6-2013

FPGA Hardware Accelerators - Case Study on Design Methodologies and Trade-Offs

Matthew V. Ryan

FPGA Hardware Accelerators - Case Study on Design Methodologies and Trade-Offs

by

Matthew V. Ryan

A Thesis Submitted in Partial Fulfillment of the Requirements for the
Degree of Master of Science
in Electrical Engineering

Supervised by
Dr. Marcin Lukowiak
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
06 / 2013

Approved by:

Dr. Marcin Lukowiak,
Department of Computer Engineering

Dr. Dorin Patru,
Department of Electrical and Microelectronic Engineering

Dr. Sonia Lopez,
Department of Computer Engineering

Dr. Sohail Dianat,
Department of Electrical and Microelectronic Engineering

Thesis Release Permission Form

Rochester Institute of Technology
Kate Gleason College of Engineering

Title:

FPGA Hardware Accelerators -
Case Study on Design Methodologies and Trade-Offs

I, Matthew V. Ryan, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or part.

Matthew V. Ryan

Date

Dedication

This thesis is dedicated to my mother and father, who have continually supported me in everything I do, regardless of where life takes me.

Acknowledgments

This thesis would not have been possible without the input and support of my advisors and various colleagues. First, I would like to thank my Masters Thesis advisors for guiding me through this process. Thanks to Dr. Lukowiak, for letting me pave my own path and try something different. Thanks to Dr. Patru and Dr. Lopez, for their advice and recommendations. Next thanks to my colleagues, who have provided countless hours of effort and support. To Chris Wood, for introducing me to HLS tools through his knowledge of Impulse C and his help with our publication. To Ganesh Khedar, for being there with me at late hours in the lab during EDA and throughout the summer and fall of performing research. Finally, thanks to Sam Skalicky, for his commitment and dedication, his somehow unlimited availability, and for all his invaluable advice.

Abstract

FPGA Hardware Accelerators - Case Study on Design Methodologies and Trade-Offs

Matthew V. Ryan

Supervised by: Dr. Marcin Lukowiak

Previous research has shown that the performance of any computation is directly related to the architecture on which it is performed. As a result, the performance of compute intensive applications can be improved using heterogeneous systems. These systems consist of various processor architectures such as CPU, FPGA, DSP, and GPU. Individual computations can be performed in parallel on different processor architecrues within the heterogeneous system. Computations are performed by utilizing existing designs from implementation libraries. There is a lack of FPGA accelerators for use in these libraries and as such additional implementations need to be designed.

Different design methodologies for developing FPGA accelerators result in implementations that vary in performance, design time, and resource utilization. A particular method and supporting toolset may produce better results for one type of design than another.

The customary method for designing FPGA accelerators is to develop the system architecture from an algorithm and model it using a hardware decription language (HDL). Another method is to convert directly from a software implementation to HDL. This process is known as high level synthesis (HLS).

The advantages and disadvantages of these two techniques can be examined through comparison of different linear algebra operations. Many linear algebra operations are parallel in nature which makes them potentially good

choices to speedup through implementation on an FPGA. In particular, matrix multiplication is an excellent candidate for examination due to not only its parallelism but also its multitude of different algorithms. The goal of this research is to design different matrix multiplication accelerators and provide insight into the advantages and disadvantages of each design procedure.

Contents

Chapter 1

Background and Motivation

1.1 Introduction

Compute intensive applications (including stock market evaluation, weather prediction, and medical diagnosis) often have impractical execution times when implemented using traditional CPUs. This leads to alternative hardware implementation in GPUs or FPGAs being required. These devices can be used alongside CPUs in order to increase performance. Individual computations can be assigned to different devices using a system scheduler controlled by a CPU. The resulting design is a heterogeneous system. An example of such a heterogeneous system is presented in Figure 1.1. The computations of these applications oftentimes consist of linear algebra operations such as matrix inverse, matrix decomposition, matrix-vector multiplication, and matrix-matrix multiplication [13].

A number of choices must be made in order to select the hardware implementation that provides the best performance. The first is the selection of the device that will perform the computation. Both GPUs and FPGAs have been shown to be suitable alternatives to CPU implementations. This is due in part to their ability to perform operations simultaneously and to more directly control the execution of operations. Additional factors also contribute to device selection including the range of the input data, the required precision, and the available memory bandwidth.

The next step is making the most efficient use of hardware resources given the chosen computational device. If an FPGA is selected a number of decisions must be made. A particular architecture must be chosen from a library along with the number of pipelines or pipeline size. Constraints

such as hardware area and available memory bandwidth of the system influence these choices. Given the variability in these factors, there is a demand for a large variety of FPGA accelerators in order to meet the performance demands of different systems. The traditional process of developing a fully custom FPGA accelerator limits the practicality of such an approach.

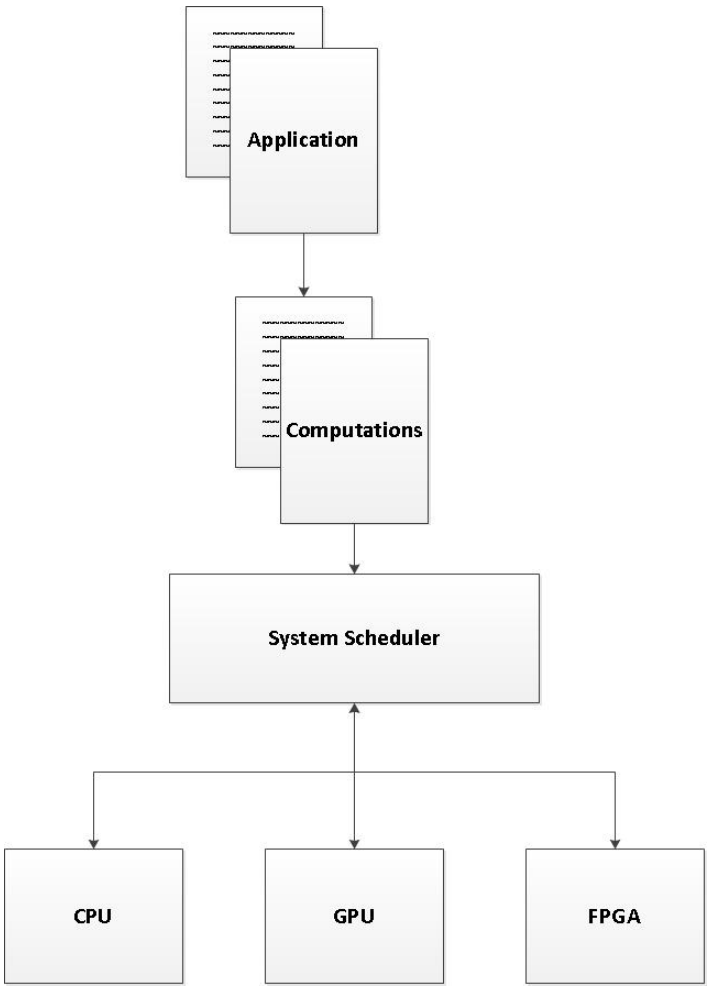

Figure 1.1: Example of a heterogeneous system utilizing CPU, GPU, and FPGA.

Two different methods for developing accelerators are using high level synthesis (HLS) tools and designing a custom implementation. High level synthesis tools convert software designs into hardware systems. Optimizations can be made within the HLS tool in order to improve the performance of the accelerator by taking advantage of the benefits of the FPGA architecture. Examples of HLS tools include Impulse C by Impulse Accelerated

Technologies and Vivado HLS which is supported by Xilinx [4]. A custom implementation is developed as a specific architecture that optimizes performance through direct control of the amount of hardware resources dedicated to the accelerator. A variety of synthesis tools exist for use in designing a custom implementation including ones supported by Xilinx, Altera, and Synopsys.

An area yet to be explored is the difference in performance, design time, and resource utilization between the two design techniques. In order to obtain realistic results, it is necessary to choose a medium for comparison. As mentioned previously, linear algebra operations consitute a large percentage of the computations within a class of applications that could benefit from implementation on a heterogeneous system. Among these, matrix-matrix multiplication stands out as a premier candidate for examination due to its exploitable parallelism and variety of different algorithms. The inherent paralellism is important because it gives incentive to implement the computation on an FPGA rather than on a CPU. The multitude of algorithms is important because implementing different algorithms provides additional information on how they vary under different circumstances.

The purpose of this work is to research new techniques of hardware development in order to improve the efficiency of accelerator design for use in hetereogeneous systems. This is accomplished through the design of three distinct matrix multiplication algorithms (standard, Strassen, and sparse matrices) using three different design techniques (software, HLS, and custom).

The goals for the software portion of this work are to design and test successful implementations of each of the three algorithms. The algorithms were implemented in C++ on an Intel Core i7 Sandy Bridge 3.4 GHz processor. The designs operated on integers for simplicity.

The design of each of the HLS implementations begins with preparing the software implementations for conversion using the HLS tool. The initial architectures of the described multipliers must then be examined. The result of testing these multipliers demonstrates the ability of the HLS tools to provide a speedup with a minimal expenditure of design time. The next step in the design is to utilize the directives within the HLS tool to take advantage of the FPGA platform and optimize the different architectures.

Many of the directives improve run time at the cost of consuming additional FPGA resources. Thus a careful balance must be struck between increasing the performance of the multiplier and straining the resources of the FPGA. The run time results are saved using the provided evaluation metrics within the HLS tools.

The custom implementation section of the work begins with researching and understanding the three designs described in the references [5], [2], and [11]. Each algorithm must be individually examined and implemented through architecture design and HDL modeling. The design of each custom implementation is modeled after was has been descibed in the background section with minor modifications. Each of the designs are developed for implementation on the target platform, the Xilinx XC6VSX475T. Every algorithm implementations is designed for operating on 32 bit precision integer operands. The run time results are determined through implementation of each custom design.

After all implementations for each algorithm are completed comparisons are made between design time and run time for each algorithm. In addition, comparisons are made between the resource consumption of the HLS implementations and the custom implementations.

Chapter 2

Supporting Work

2.1 FPGA Overview

FPGAs consist of a set of reconfigurable resources that can be configured to implement particular function. The resources consist of configurable logic blocks (CLBs), input-output buffers (IOBs), digital clock managers (DCMs), digital signal processor slices (DSPs), and block rams (BRAMs). A high level overview of an FPGA is presented in Figure 2.1 [6]. Figure 2.2 shows the contents of an FPGA configurable logic block [8]. The components of an FPGA slice are presented in Figure 2.3 [8].

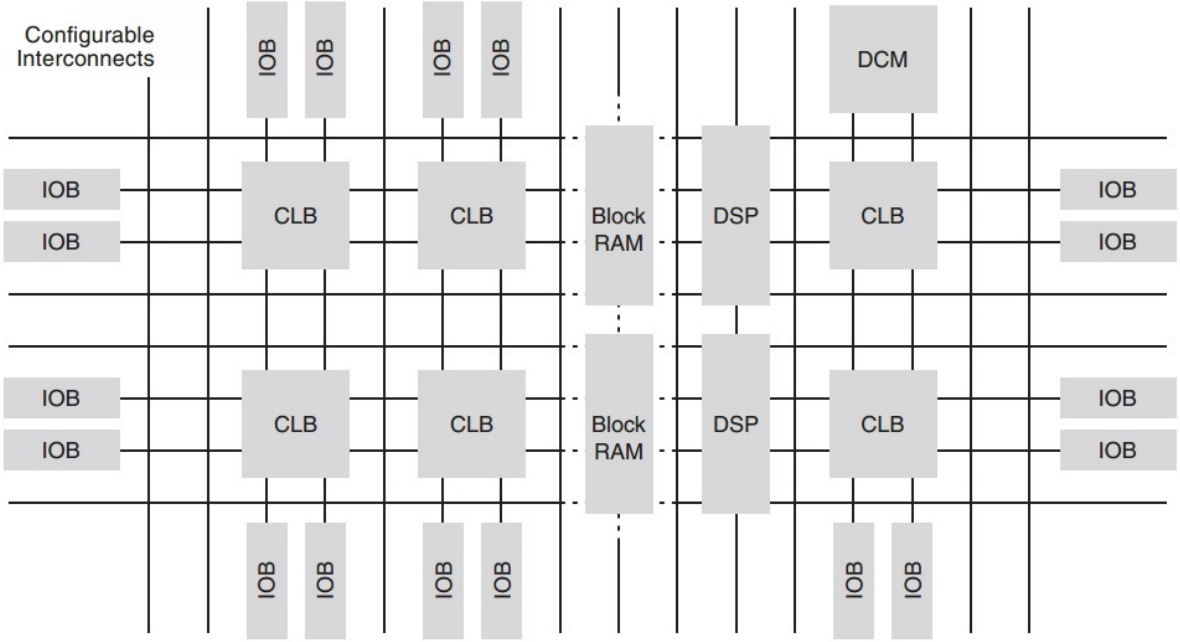

Figure 2.1: Example set of reconfigurable resources available on a Xilinx FPGA [6].

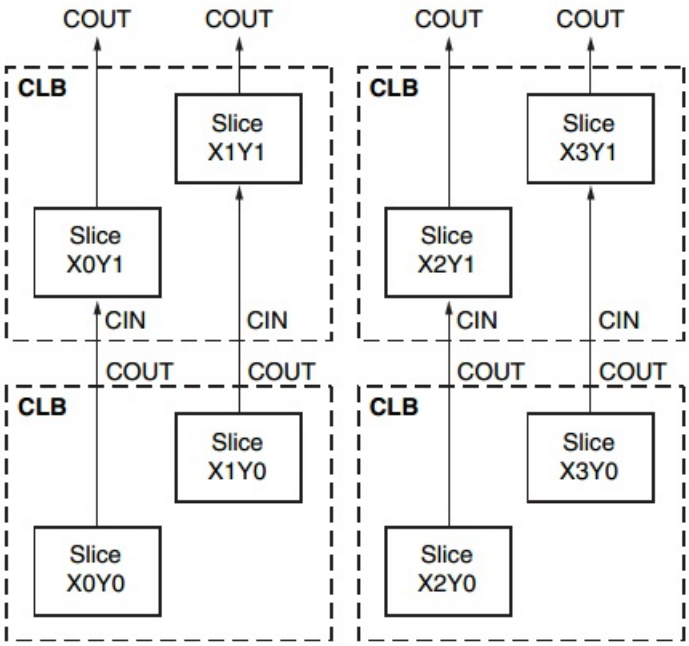

Figure 2.2: Example contents of a configurable logic block within a Xilinx FPGA [8].

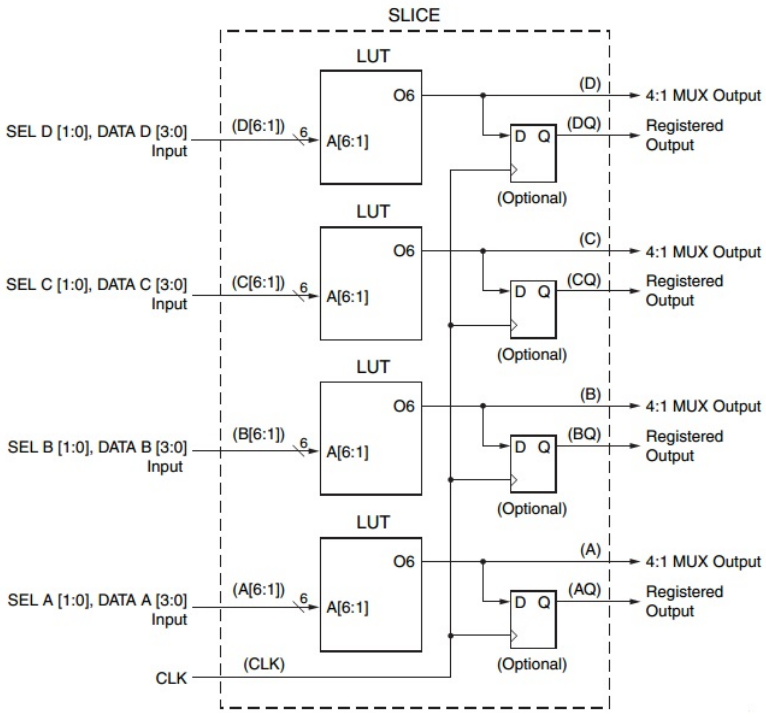

Figure 2.3: Example contents of a slice on a Xilinx FPGA [8].

FPGAs are very efficient for use in use in digital signal processing applications due to their highly parallel nature and ability to implement custom algorithms. Applications that require many binary multipliers and adders are best implemented using dedicated DSP slices. DSP slices contain built-in cascade logic that allows multiple DSP slices to be connected together in order to implement complex functions. Without this ability the FPGA would have to develop large and inefficient adder trees to implement this functionality. A diagram demonstrating the basic functionality of the DSP slices present in the Virtex 6 is presented in Figure 2.4 [9].

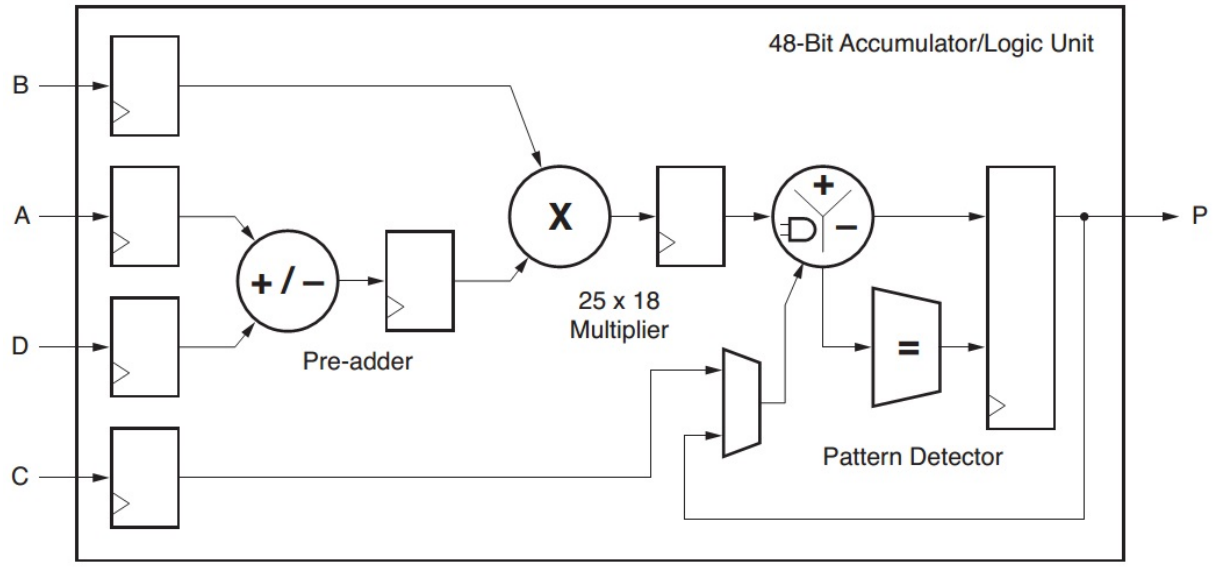

Figure 2.4: Example architecture of DSP slice on a Xilinx FPGA [9].

Xilinx has its own line of memory solutions that provide the interface between user generated designs and off chip memory components. The physical layer of the design connects to to the memory device via the on-board FPGA IOBs. The user interface is connected within the FPGA logic. Figure 2.5 shows the memory interface solution (MIS) [10].

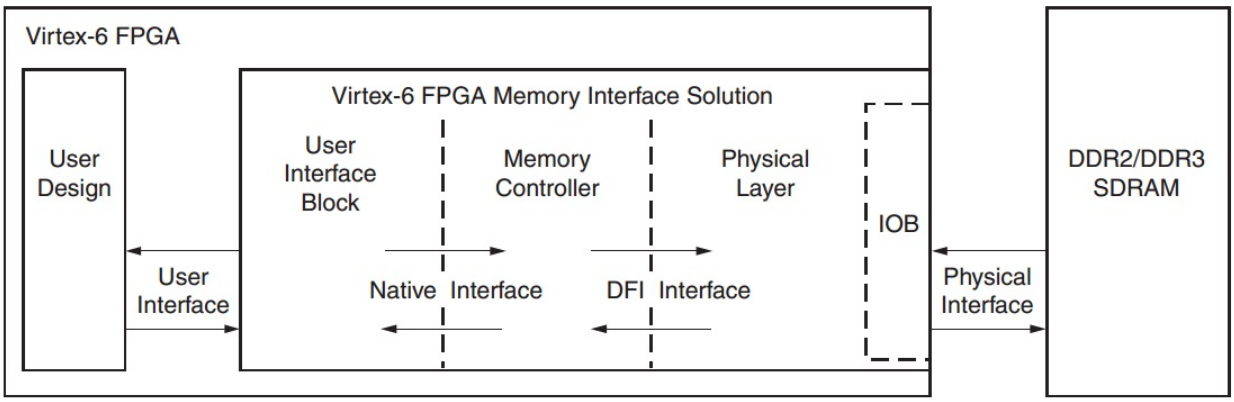

Figure 2.5: Architecture of the Xilinx memory interface [10].

The user interface block provides a simple interface to the memory component from the user logic. It also buffers all read and write data. In addition, it reorders the read return data to match the request order and presents a flat address space to the user that it translates to the address space required by the memory.

The memory controller block receives the requests from the user design and reorders them to minimize stall states. This feature serves to increase the performance of the memory component. It also performs high level management functions such as refresh and activate/precharge.

The physical block interfaces with the memory controller block and translates the internal signals into the actual signals that connect to the memory component. This block also synchronizes the control signals and data over the various clock domains. In additon, it also performs the necessary initialization and management of the memory device.

2.2 Design Methodologies

2.3 Custom Design Flow

Figure 2.6 displays the design flow for a custom hardware design [7].

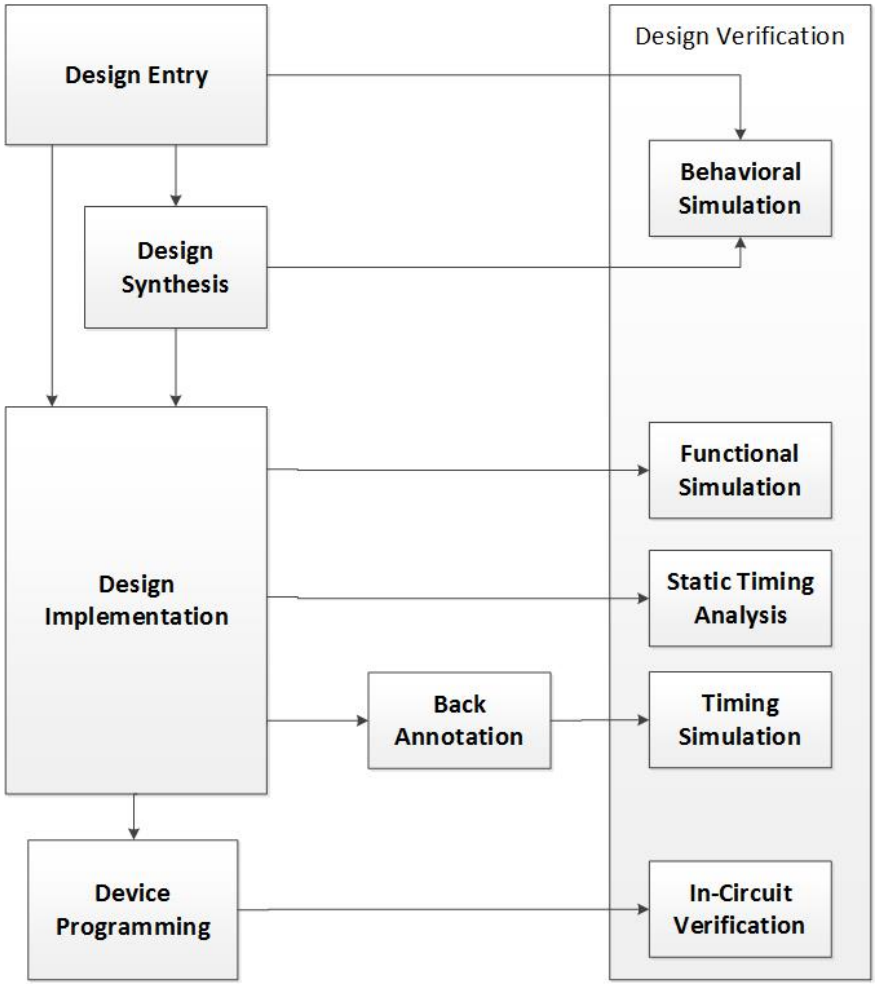

Figure 2.6: Example flow for custom FPGA design using traditional hardware description languages [7].

The design is first implemented using a hardware description language (HDL). Each sub-component within the design is tested for proper functionality using a behavioral simulation. After the full design is complete it is synthesized and a final behavioral simulation is performed. The next step in the process is implementation. The implementation stage includes mapping the design to the target device, placement of the design within the device, routing of the custom logic, and ultimately bitstream generation. Throughout this process the design undergoes numerous levels of testing. The first is a functionality simulation, which tests the basic functionality of the design. Additionally, static timing is performed which determines the

necessary timing contraints of the implementation. Finally, a timing simulation is performed which evaluates the design with all timing constraints implemented. After design implementation the FPGA is programmed with the resulting bitstream and on-chip verification is performed.

2.4 HLS Design Flow

The design flow for an HLS design is presented in Figure 2.7 [4].

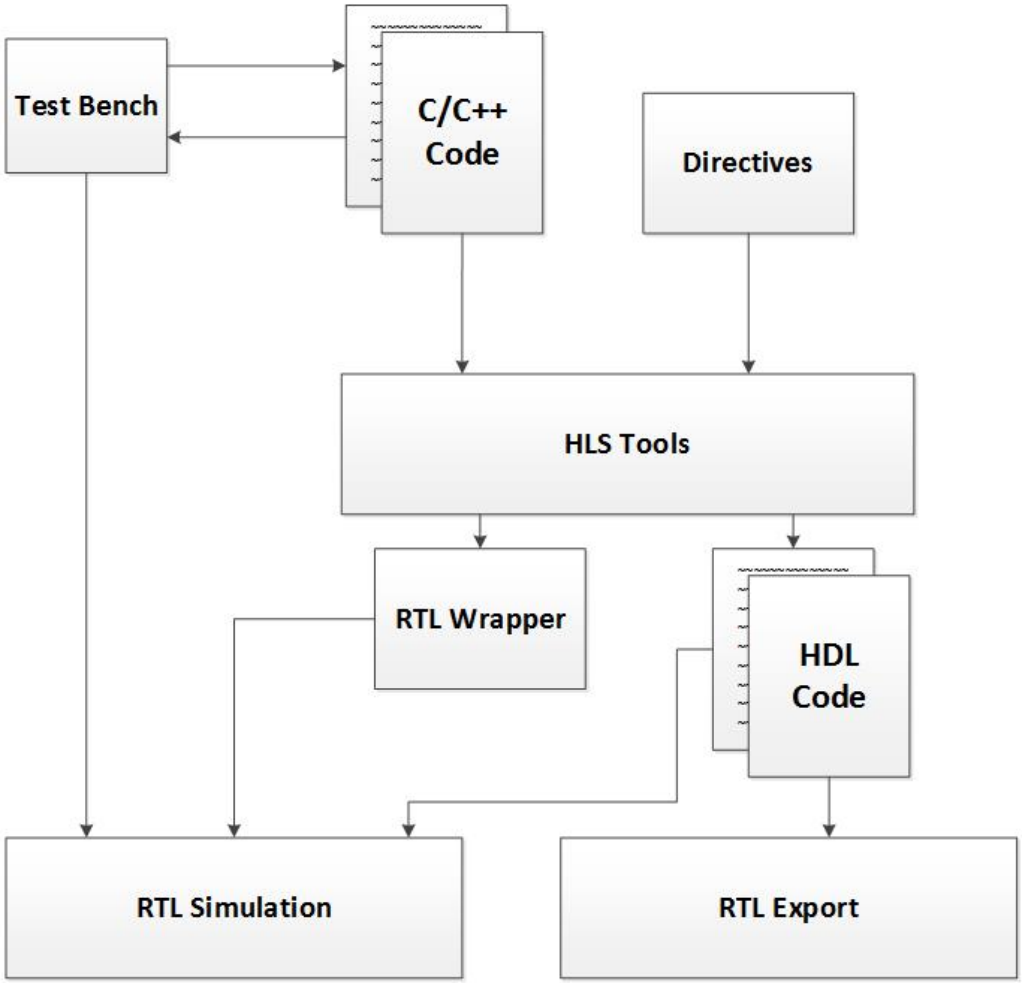

Figure 2.7: Example flow for FPGA design using the Vivado HLS tool [4].

The design for an HLS implementation begins with source code program a programming language (such as C or C++) that is independently verified

as functional. From this point the code is imported to the HLS tools. Optionally, directives can be added which can alter performance and resource consumption. Directives will be discussed in more detail further in the document. A register-transistor logic (RTL) wrapper is developed using the HLS tools which can be used to verify the design. Once the design is successfully verified it can be packaged and exported in a convenient fashion for use in an exisiting system.

2.5 Matrix Multiplication Algorithms

2.6 Standard Algorithm

The standard algorithm for matrix-matrix multiplication multiplies each element of each row in input matrix A with each element of each column in input matrix B [5]. The results of each row/column combination are summed, whichs results in an element of output matrix C. The algorithm is demonstrated in Figure 2.8. This particular algorithm requires $n \times m \times p$ elementary multiplications and additions, where m and n are the number of rows and columns in matrix A and n and p are the number of rows and columns in matrix B. For the special case in which both A and B are square matrices, the number of additions and multiplications are both equal to N^3, where N is the number of rows and columns in both A and B.

$$A = \begin{bmatrix} a_{1,1} & a_{12} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \dots & a_{m,n} \end{bmatrix} B = \begin{bmatrix} b_{1,1} & b_{1,2} & \dots & b_{1,p} \\ b_{2,1} & b_{2,2} & \dots & b_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n,1} & b_{n,2} & \dots & b_{n,p} \end{bmatrix} C = \begin{bmatrix} c_{1,1} & c_{1,2} & \dots & c_{1,p} \\ c_{2,1} & c_{2,2} & \dots & c_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m,1} & c_{m,2} & \dots & c_{m,p} \end{bmatrix}$$

$$C = A \times B$$

$$c_{i,j} = \sum_{k=1}^{n} (a_{i,k} \times b_{k,j})$$

Figure 2.8: General description of the standard matrix multiplication algorithm.

2.7 Block Multiplication

Block based multiplication is a method of matrix-matrix multiplication that is particularly useful for parallel based implementations. In order to perform this method of multiplication, it is necessary to partition the source matrices into separate smaller matrices called blocks. Figure 2.9 shows a matrix P with 6 rows (m) and 6 columns (n) of elements [12]. Figure 2.10 shows the process of partitioning matrix P into blocks. The block extends until it reaches a limit of elements defined by BB, the basic block size. In this example, $BB = 3$.

Figure 2.11 shows the procedure of performing block multiplication. The resulting matrix C is developed from performing operations on blocks as opposed to individual elements.

$$P = \begin{bmatrix} p_{1,1} & p_{1,2} & p_{1,3} & p_{1,4} & p_{1,5} & p_{1,6} \\ p_{2,1} & p_{2,2} & p_{2,3} & p_{2,4} & p_{2,5} & p_{2,6} \\ p_{3,1} & p_{3,2} & p_{3,3} & p_{3,4} & p_{3,5} & p_{3,6} \\ p_{4,1} & p_{4,2} & p_{4,3} & p_{4,4} & p_{4,5} & p_{4,6} \\ p_{5,1} & p_{5,2} & p_{5,3} & p_{5,4} & p_{5,5} & p_{5,6} \\ p_{6,1} & p_{6,2} & p_{6,3} & p_{6,4} & p_{6,5} & p_{6,6} \end{bmatrix}$$

Figure 2.9: Matrix P

$$P_{11} = \begin{bmatrix} p_{1,1} & p_{1,2} & p_{1,3} \\ p_{2,1} & p_{2,2} & p_{2,3} \\ p_{3,1} & p_{3,2} & p_{3,3} \end{bmatrix} P_{12} = \begin{bmatrix} p_{1,4} & p_{1,5} & p_{1,6} \\ p_{2,4} & p_{2,5} & p_{2,6} \\ p_{3,4} & p_{3,5} & p_{3,6} \end{bmatrix}$$

$$P_{21} = \begin{bmatrix} p_{4,1} & p_{4,2} & p_{4,3} \\ p_{5,1} & p_{5,2} & p_{5,3} \\ p_{6,1} & p_{6,2} & p_{6,3} \end{bmatrix} P_{22} = \begin{bmatrix} p_{4,4} & p_{4,5} & p_{4,6} \\ p_{5,4} & p_{5,5} & p_{5,6} \\ p_{6,4} & p_{6,5} & p_{6,6} \end{bmatrix}$$

$$P = \begin{bmatrix} P_{11} & P_{12} \\ P_{21} & P_{22} \end{bmatrix}$$

Figure 2.10: Example of block partitioning with BB=3 and N=6.

$$C = A \times B$$

$$\begin{bmatrix} C_{11} & C_{1N} \\ C_{N1} & C_{NN} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{1N} \\ A_{N1} & A_{NN} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{1N} \\ B_{N1} & B_{NN} \end{bmatrix}$$

$$\mathrm{C}_{ij} = \sum_{k=1}^{N} A_{ik} \times B_{kj}$$

Figure 2.11: Example of matrix block multiplication.

2.8 Strassen Algorithm

The Strassen algorithm operates on 2×2 matrices and is designed to reduce the number of multiplications operations at the expense of requiring additional summations [2]. Intermediary results $s_1 - s_7$ are defined as functions of the input elements $a_{11} - a_{22}$ and $b_{11} - b_{22}$. The output results $c_{11} - c_{22}$ are defined as additions/subtractions of the intermediary s results [3]. An overview of the Strassen algorithm is presented in Figure 2.12. Only 7 multiplications are required in order to complete the operation. This is in contrast to the standard algorithm, which would require $N^3 = 8$ multiplications in order to obtain the same result. However, 18 additions/subtractions are required for the Strassen algorithm to complete the computation, whereas only 8 are required for the standard algorithm. In order to handle matrix multiplications of a larger size, a block base approach as described above is used.

$$C = A \times B$$

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

$$\begin{aligned} \mathrm{s}_1 &= (\mathrm{a}_{11} + a_{22}) \times (b_{11} + b_{22}) \\ \mathrm{s}_2 &= (\mathrm{a}_{21} + a_{22}) \times b_{11} \\ \mathrm{s}_3 &= \mathrm{A}_{11} \times (b_{12} - b_{22}) \\ \mathrm{s}_4 &= \mathrm{A}_{22} \times (b_{21} - b_{12}) \\ \mathrm{s}_5 &= (\mathrm{a}_{11} + a_{12}) \times b_{22} \\ \mathrm{s}_6 &= (\mathrm{a}_{21} - a_{11}) \times (b_{11} + b_{12}) \\ \mathrm{s}_7 &= (\mathrm{a}_{12} - a_{22}) \times (b_{21} + b_{22}) \\ \mathrm{c}_{11} &= \mathrm{s}_1 + s_4 - s_5 + s_7 \\ \mathrm{c}_{12} &= \mathrm{s}_3 + s_5 \\ \mathrm{c}_{21} &= \mathrm{s}_2 + s_4 \\ \mathrm{c}_{22} &= \mathrm{s}_1 - s_2 + s_3 + s_6 \end{aligned}$$

Figure 2.12: General description of the Strassen matrix multiplication algorithm.

2.9 Sparse Matrices Algorithm

Both of the aforementioned algorithms have assumed that the source matrices are dense. When matrices consist largely of zero value elements it is possible to compact the sparse matrix into a form in which its sparsity can be easily exploited. In this work, sparse matrices are stored in the compressed sparse row (CSR) and compressed sparse column (CSC) formats. A sparse matrix displayed in CSR format is comprised of three vectors as shown in Figure 2.13. The first vector, val, consists of the values of the non-zero elements of the sparse matrix. The second, col, contains the column index of each of the non-zero elements of the sparse matrix. Finally, row stores the index in val of the first non-zero element of row i. Conversely, CSC format stores the row index of each non-zero element in the row vector and the index of the first non-zero element of each column in the col vector [1].

$$S = \begin{bmatrix} 0 & s_{1,2} & 0 & 0 \\ 0 & s_{2,2} & s_{2,3} & 0 \\ s_{3,1} & 0 & 0 & s_{3,4} \\ s_{4,1} & 0 & 0 & 0 \end{bmatrix}$$

val	$s_{1,2}$	$s_{2,2}$	$s_{2,3}$	$s_{3,1}$	$s_{3,4}$	$s_{4,1}$
col	1	1	2	0	3	1
row	0	1	3	5		

val	$s_{3,1}$	$s_{4,1}$	$s_{1,2}$	$s_{2,2}$	$s_{2,3}$	$s_{3,4}$
row	2	3	0	1	1	2
col	0	2	4	5		

Figure 2.13: Example sparse matrix (top) compressed in CSR (middle) and CSC (bottom) formats.

2.10 HLS

HLS is a fairly new form of accelerator development that converts C and C++ software into a hardware design. HLS tools have numerous means available which allow for adjusting the architecture of the algorithms for the FPGA platform. The primary method of improving performance is to apply directives to a design. Directives are commands that instruct the HLS tool to implement special functions to an HLS Design. One such directive is loop pipelining. When used on a loop within the HLS tool, the pipeling directive allows different loop iterations to overlap in time. Figure 2.14 shows a simple loop that performs three different operations. Table 2.1 shows how the loop would be executed with no directives (architecture control). Table 2.2 displays the execution of the loop after applying the pipelining directive [4].

Clock Cycle	1	2	3	4	5	6
Operation	*read_op*	*compute_op*	*write_op*	*read_op*	*compute_op*	*write_op*

Table 2.1: Example loop execution (no architecture control).

```
void function(...)
  {
    for(i=0;i<=1;i++)
      {
        read_op;
        compute_op;
        write_op;
      }
  }
```

Figure 2.14: Example loop to be pipelined.

Clock Cycle	1	2	3	4
Operation	*read_op*	*compute_op*	*write_op*	
Operation		*read_op*	*compute_op*	*write_op*

Table 2.2: Example loop execution (pipelined).

Another example of an HLS directive is loop-unrolling. Loop-unrolling separates for-loops into multiple independent operations rather than a single group of operations. Loops can be unrolled fully or partially. Figure 2.15 shows a multiplication operation performed over 4 iterations of a for-loop. Table 2.3 shows how the loop would be executed with no architecure control. Table 2.4 displays the execution of the loop after applying the unroll directive with a factor of 2. Table 2.5 shows the loop execution after fully unrolling it [4].

```
void function(...)
  {
    for(i=0;i<=3;i++)
      {
        C[i] = A[i] * B[i];
      }
  }
```

Figure 2.15: Example loop to be unrolled.

$Clock\ Cycle$	1	2	3	4
$Operation$	$Read\ A[0]$	$Read\ A[1]$	$Read\ A[2]$	$Read\ A[3]$
$Operation$	$Read\ B[0]$	$Read\ B[1]$	$Read\ B[2]$	$Read\ B[3]$
$Operation$	$*$	$*$	$*$	$*$
$Operation$	$Write\ C[0]$	$Write\ C[1]$	$Write\ C[2]$	$Write\ C[3]$

Table 2.3: Example Loop Execution (No Architecture Control)

$Clock\ Cycle$	1	2
$Operation$	$Read\ A[0]$	$Read\ A[2]$
$Operation$	$Read\ B[0]$	$Read\ B[2]$
$Operation$	$Read\ A[1]$	$Read\ A[2]$
$Operation$	$Read\ B[1]$	$Read\ B[2]$
$Operation$	$*$	$*$
$Operation$	$*$	$*$
$Operation$	$Write\ C[0]$	$Write\ C[2]$
$Operation$	$Write\ C[1]$	$Write\ C[3]$

Table 2.4: Example loop execution (unrolled factor = 2)

$Clock\ Cycle$	1
$Operation$	$Read\ A[0]$
$Operation$	$Read\ B[0]$
$Operation$	$Read\ A[1]$
$Operation$	$Read\ B[1]$
$Operation$	$Read\ A[2]$
$Operation$	$Read\ B[2]$
$Operation$	$Read\ A[3]$
$Operation$	$Read\ B[3]$
$Operation$	$*$
$Operation$	$*$
$Operation$	$*$
$Operation$	$*$
$Operation$	$*$
$Operation$	$Write\ C[0]$
$Operation$	$Write\ C[1]$
$Operation$	$Write\ C[2]$
$Operation$	$Write\ C[3]$

Table 2.5: Example loop execution (fully unrolled).

Both pipelining and loop unrolling reduce the run times of matrix multiplication computations. However, these improvements also increase the

number of hardware components necessary for the HLS design. This in turn can reduce the maximum operating clock frequency by creating a larger longer critical path through the design, reducing performance.

2.11 Custom

An interesting custom implementation of the standard algorithm has been studied in [5]. In said work matrix-matrix multiplication is identified as a major bottleneck in facial recognition systems. According to the research, in a sample of facial recognition algorithms examined over eighty percent of the computation time was spent on matrix multiplication [5].

The technology of choice for this work was the Virtex 5 VSX240T. The reference architecture was designed to perform the two innermost for-loops of the standard algorithm in parallel. This means that $N \times N$ multiplications and additions were performed simultaneously. However, as the matrix multiplication was performed on a block by block basis, N in this case did not refer to the size of a source matrix, but rather the size of the matrix block that is being computed. As such, this value is referred to as the basic block size (BB) and N maintains its original meaning as the size of an input matrix. In their work $BB = 16$, meaning that $BB^2 = 16^2 = 256$ elementary multiplications and additions were performed simultaneously. Thus this implementation performs the standard algorithm by partitioning the input matrices into blocks of sixteen elements and then repeatedly performing calculations until the full matrix computation is complete. The result is a matrix multiplication computation that was claimed to be more than fourty times faster than similar systems implemented previously on reconfigurable devices. Table 2.6 shows the experimental results from [5]. Table 2.16 shows an example implementation with $N = 2$ and $BB = 2$.

Several different variants on FPGA implementations of the Strassen algorithm have been studied in [2]. The design with the highest performance was one in which the input matrices were broken down into block matrices of size 2×2. The technology chosen for this work was the Xilinx XC2V500-FG256-5. A custom 2×2 Strassen multiplier was developed

$Matrix\ Dimensions$	$Execution\ Time\ (ms)$
$(64, 64) \times (64, 64)$	0.022
$(128, 128) \times (128, 128)$	0.071
$(256, 256) \times (256, 256)$	0.454
$(512, 512) \times (512, 512)$	3.645
$(1024, 1024) \times (1024, 1024)$	29.063

Table 2.6: Results from [5].

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$$

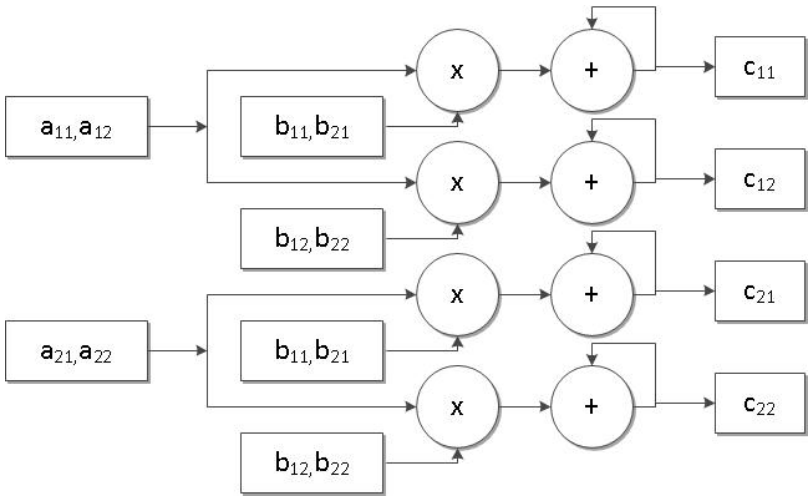

Figure 2.16: Standard algorithm implementation with N=2 and BB=2.

and used to compute the blocks of the result matrix. Several of these multipliers were used simultaneously in order to speedup the run time of the matrix multiplication computation. The number of 2×2 multipliers used is the basic element (BE) count.

The paper compares the results of implementing the designs with regards to two factors: computation run time and FPGA slices consumed. The test matrix sizes were 8×8, 32×32, 64×64, 256×256, and 512×512. The results showed that the described implementation consumed half as many slices as its closest competitor for all matrix sizes tested. In addition, it equalled the run time of the fastest design for the entire range of data sizes. Table 2.7 shows the experimental results from [2]. Figure 2.17 shows the basic element and Figure 2.18 shows an example implementation with

$BE = 4$ and $N = 4$.

$Matrix\ Dimensions$	$Execution\ Time\ (ms)$
$(8,8) \times (8,8)$	0.035
$(32,32) \times (32,32)$	0.120
$(64,64) \times (64,64)$	1.523
$(256,256) \times (256,256)$	100.562
$(512,512) \times (512,512)$	945.312

Table 2.7: Results from [2].

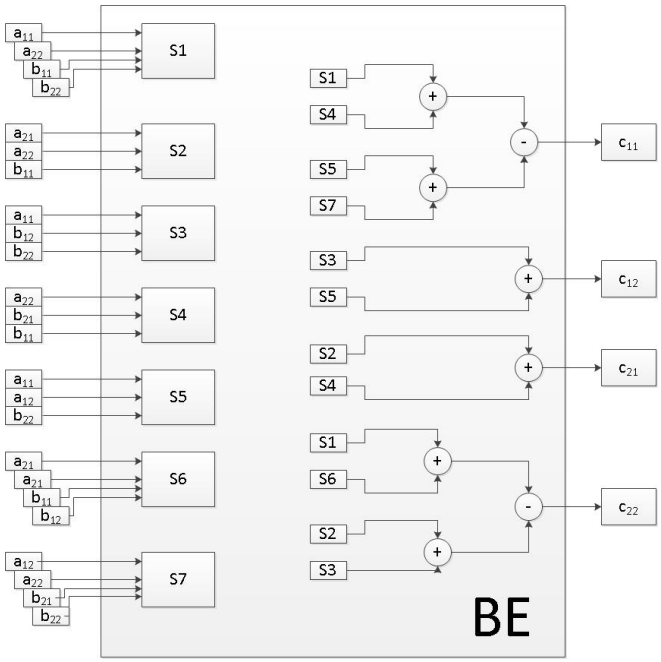

Figure 2.17: Design of Strassen Basic Element (BE).

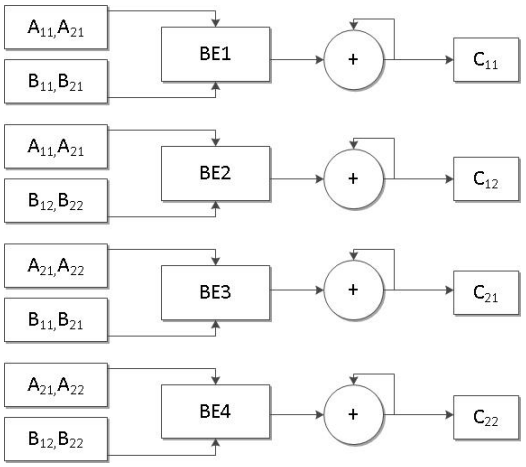

Figure 2.18: Example of Strassen implementation with BE=4 and N=4.

Some work has been presented on sparse matrix multiplication implemented on FPGAs. A design of interest is that presented in [11]. The Xilinx XC5VLX110T FPGA was the technology utilized for this work. The chosen architecture for this particular implementation is that of a systolic array. The systolic array consists of processing elements (PEs) that pass data back and forth between one another in order to keep off-chip memory accesses to a minimum. The PE is defined as a multiply-accumulator, three memory elements, registers, and control logic. Like the other custom implementations, this design relies on block-based multiplication in order to perform large matrix-matrix multiplication computations. The focus of this work is on balancing the power-delay product and the energy-delay product. The power-delay product is used to estimate the tradeoff between energy consumption and delay. The energy-delay product indicates the tradeoff between performance (run time) and energy consumption of the system.

This work found that there were two defining parameters of importance when designing the sparse matrix multiplier. These were the number of PEs and the choice in matrix block size. In order to evaluate the performance of the various designs, two metrics were used: the power-delay product and the energy-delay product. The power-delay product was the power consumption of a design multiplied by its computational delay. The energy-delay product was the energy consumed by the design multiplied by its computational delay. This work found that a better power-delay product

was achieved when utilizing a smaller number of PEs and a smaller block size. Contrarily, a better energy-delay product is obtained by using a large number of PEs and a large basic block size. Tables 2.8 - 2.11 display the experimental results presented in [11]. Figure 3.4 shows the design of the sparse processing element. Figure 3.5 shows an example implementation with a variable number of processing elements.

$Number\ of\ PEs$	$Density = 100$	30	20	10
4	30.5	38.1	39.5	48.2
8	18.0	25.0	28.1	37.3
16	13.2	24.8	29.1	29.5
32	10.0	22.1	29.5	52.1
64	9.4	26.2	37.5	80.8

Table 2.8: Power-delay product (in mW$\times$cycles/operation) versus number of PEs.

$Number\ of\ PEs$	$Density = 100$	30	20	10
4	7.90	11.80	13.6	18.1
8	2.10	4.62	5.90	9.90
16	1.00	3.10	4.00	9.50
32	0.33	2.00	3.52	9.80
64	0.12	1.90	3.82	16.02

Table 2.9: Energy-delay product (in mJ$\times$cycles/operation) versus number of PEs.

$Block\ Size$	$Density = 100$	30	20	10
96	7.50	19.50	26.11	54.30
128	7.41	16.23	21.12	39.19
192	10.00	19.94	31.12	39.17
256	13.21	25.02	29.82	44.32
384	24.13	39.98	45.14	64.21

Table 2.10: Power-delay (in mW$\times$cycles/operation) versus block size.

Block Size	*Density* = 100	30	20	10
4	0.51	4.11	7.11	24.96
8	0.48	2.51	4.92	14.21
16	0.51	2.54	4.56	9.93
32	0.63	3.12	4.71	9.84
64	2.12	4.95	4.08	10.44

Table 2.11: Energy-delay (in mJ×cycles/operation) versus block size.

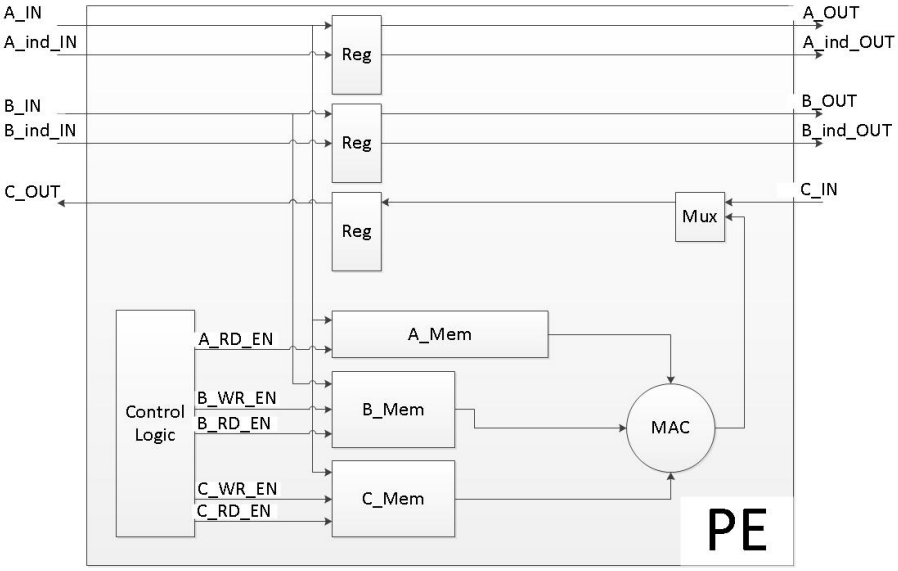

Figure 2.19: Design of sparse matrices Processing Element (PE).

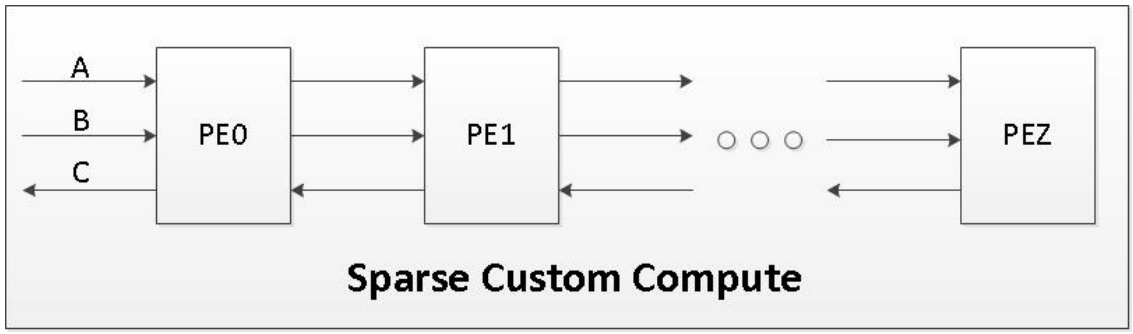

Figure 2.20: Top level sparse matrices implementation.

Chapter 3

Custom Implementations

3.1 Standard Implementation

The standard custom design was implemented using a block-based multiplication technique in an effort to best take advantage of the FPGA hardware. The product of large matrices is obtained through performing the multiplication and summation the blocks they are composed of. The size of these blocks is designated by BB. The standard algorithm is performed on the blocks until the final result for the operation is acquired. The number of iterations through the matrix multiplier necessary to compute the final result is represented by $\lceil (N/BB)^3 \rceil$ where N is the source matrix size. For this particular design, the decision was made to perform the two innermost for-loops of the standard algorithm in parallel. This meant that the number of simultaneous operations performed was equal to BB^2. Thus a small increase in basic block size resulted in a large increase in resources consumed. In this implementation the basic block size was chosen to be 16, meaning that 256 simultaneous multiplications and additions were performed.

In this design an operand of a row in input matrix A was multiplied with each operand of a row in matrix B. This technique is beneficial as it successfully saturated all 256 elementary multiplication components with only $2 \times BB = 2 \times 16 = 32$ operands. Figure 3.1 demonstrates this method of data sourcing and further clarifies the design of the multiplier compute logic.

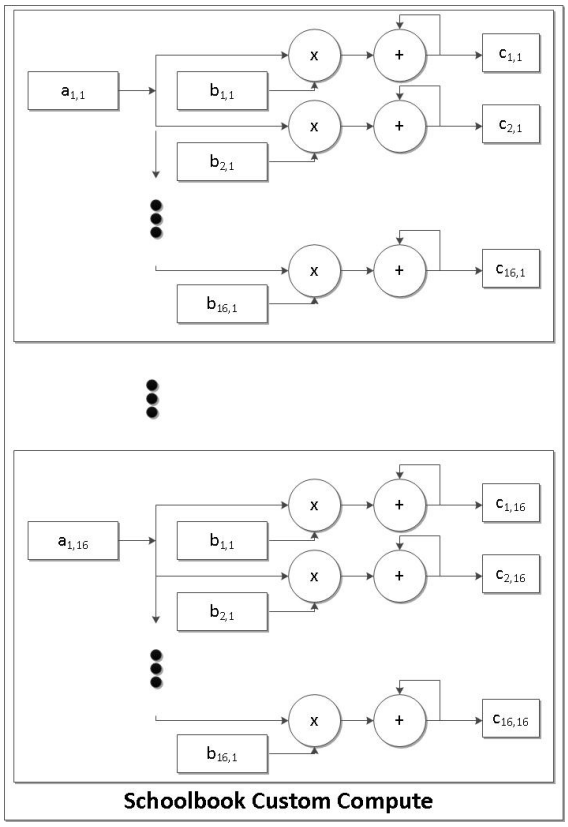

Figure 3.1: Custom standard matrix multiplier compute logic with BB=16.

3.2 Strassen Implementation

The Strassen custom implementation design began with the designation of a basic element, BE, as a 2×2 multiplier as depicted in Figure 3.2.

The BE calculates the intermediary matrices $S1-S7$ as described in the Strassen algorithm. These matrices are then then summed in order to produce the resulting output matrix. This design connected four BEs in parallel in order to more efficiently complete a 4×4 matrix multiplication. Computing a 4×4 operation using only a single BE would take $(4^3)/(2^3) = 8$ iterations. Given that four multipliers were used in parallel, only $8/4 = 2$ iterations were required for this design to complete a 4×4 matrix multiplication. Control logic determined the flow of data into the BEs. The design of the 4×4 Strassen custom multiplier is presented in Figure 3.3.

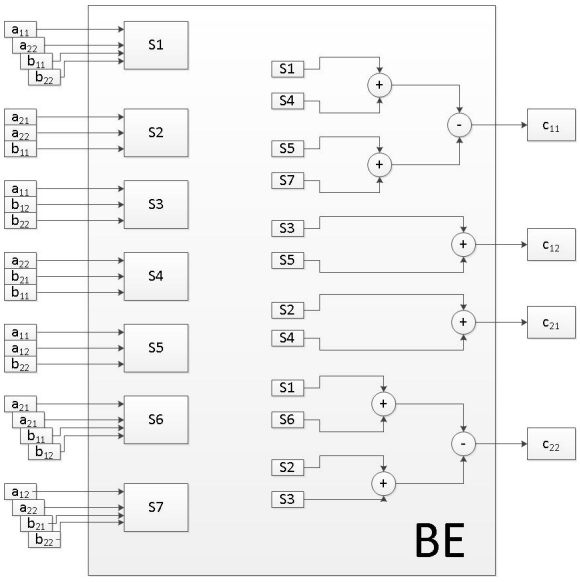

Figure 3.2: Design of Strassen basic element.

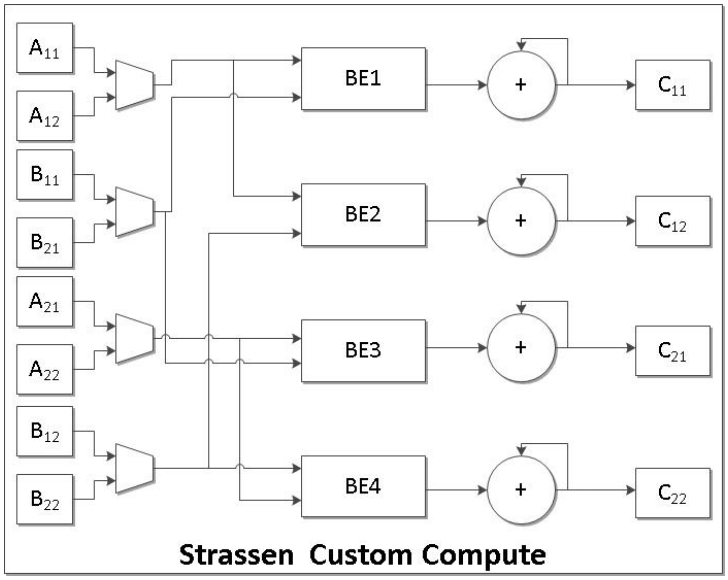

Figure 3.3: Custom Strassen compute logic with BE=4.

3.3 Sparse Implementation

The custom sparse implementation was designed as a systolic array of processing elements (PE) that operated on the non-zero operands of a sparse matrix . The design of the processing element is presented in Figure 3.4.

The PE contained control logic which determined which operands were written to memory based on their row and column indices. Each PE contained an elementary multiplier and accumulator. In addition, data was passed from one PE to the next sequential PE. In this way the workload was evenly distrubuted across all PEs with only the first PE in the array communicating with the remainder of the system. The design of the sparse compute logic is presented in Figure 3.5.

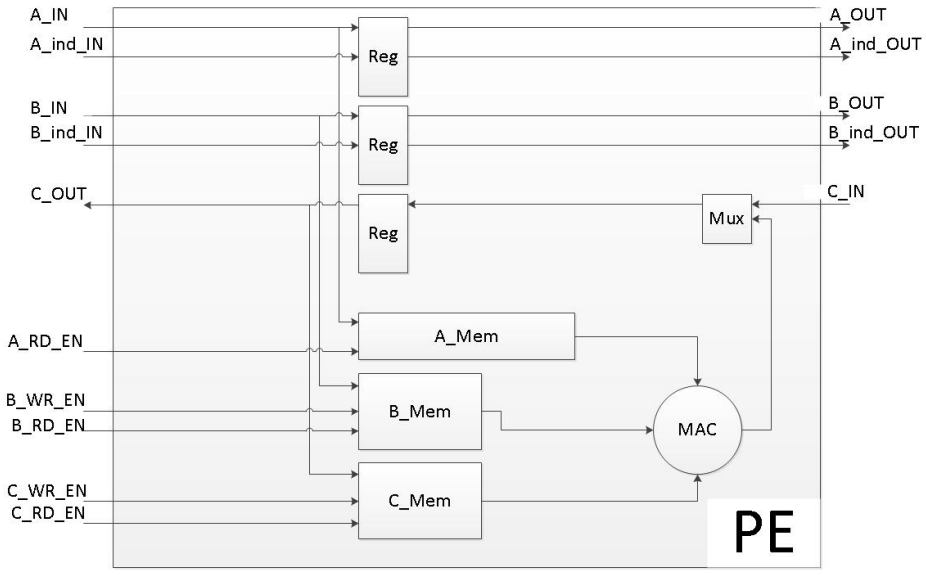

Figure 3.4: Design of sparse matrices Processing Element (PE).

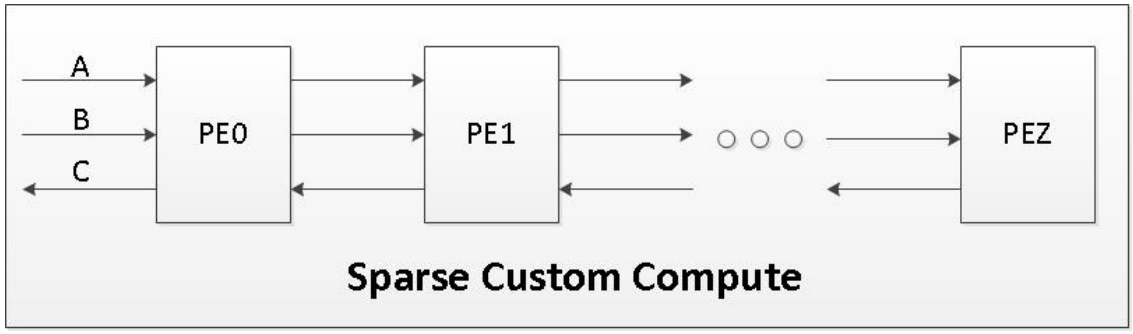

Figure 3.5: Custom sparse matrices compute logic.

Chapter 4

HLS Implementations

4.1 Standard Implementation

Figure 4.1 shows the standard multiplication algorithm as it was implemented using the HLS tools.

```
for i = 0 → rows(A) do                          ▷ Rows
    for j = 0 → cols(B) do                        ▷ Cols
        for k = 0 → rows(B) do                      ▷ Product
            C_{i,j} = C_{i,j} + A_{i,k} × B_{k,j}     ▷ Calculation
        end for
    end for
end for
```

Figure 4.1: Standard matrix multiply algorithm implementation.

The source code consisted of three nested for-loops. In order to easily distinguish between the loops they were each assigned a label: $Rows$, $Cols$, and $Product$ respectively. This is important because the architecture control within the Vivado HLS tools work by modifying individual loops. As it stands, the exact code presented in Figure 4.1 was processed through Vivado and then exported to Xilinx Design Suite in order to obtain timing and resource consuption information. Though Vivado does provide an estimate of resource usage after synthesizing a design it is not as accurate as running a full place and route in Design Suite. The matrix row and column sizes were chosen to be 16 in order to establish a basis for comparison between the different designs. The next goal was to improve the performance of the developed accelerator by applying architecure control with Vivado. For

this particular design loop-unrolling and pipelining were chosen as modifications to be made. Loop-unrolling is the premier choice for a design that features nested for-loops as it separates the loops into separate operations that can be performed independently. The pipelining directive adds efficiency by adding registers which are used to more efficiently load data into the design. It should also be noted that pipelining a top-level for-loop unrolls all for-loops nested within the top-level loop. As previously mentioned, these directives are applied on a loop-by-loop basis which allows easy manipulation of FPGA resource consumption. For loop unrolling in particular, an additional option exists (called factor) that allows partial unrolling of a loop, which can further preserve resources. It is also important to keep in mind that when a top-level for-loop is unrolled all loops within the for-loop are also unrolled. With these ideas in mind, various versions of the HLS design were implemented. First, the pipeline directive was applied to the $Product$ loop. This was followed by pipelining the $Cols$ and $Rows$ loops. The next step was to apply various levels of loop-unrolling to the accelerator. The initial loop to be unrolled was the $Product$ loop. This was followed by the unrolling of the $Cols$ loop. Both of these loops were fully unrolled. However, due to the limited number of DSP slices on the FPGA the third loop, $Rows$, could not be fully unrolled. As such, it was only partially unrolled with factors of 2 and 4. Figures 4.2 - 4.3 show examples of the RTL generated from a couple of the different variations Standard HLS design. The number of multiplexers, shift registers, multipliers, and adders varied depending on the directives applied to the design. Table 4.1 demonstrates the impact that the different directives had on the design of the hardware accelerator.

The architecture generated for the no directives implementation is a simple multiply accumulate (MAC) unit. The control logic is not shown for clarity. Unrolling the $Product$ loop only produced two multipliers sharing a single adder. However, it was expected that 8 multipliers and adders would have been generated. Due to the design of the algorithms used within the HLS tool only two multipliers with a shared adder were generated. Pipelining the $Cols$ loop also unrolled the $Product$ loop, but added registers between the various components as shown in Figure 4.3. After unrolling the

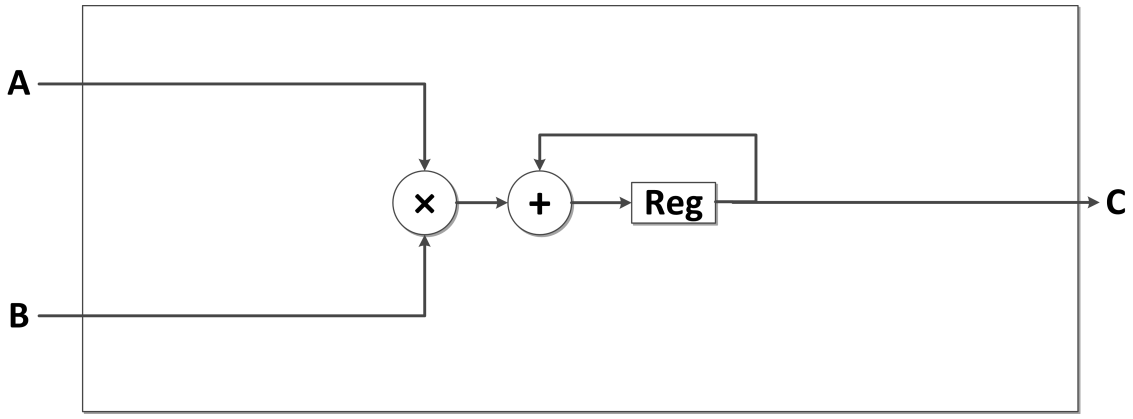

Figure 4.2: Standard HLS compute logic: no architecture control.

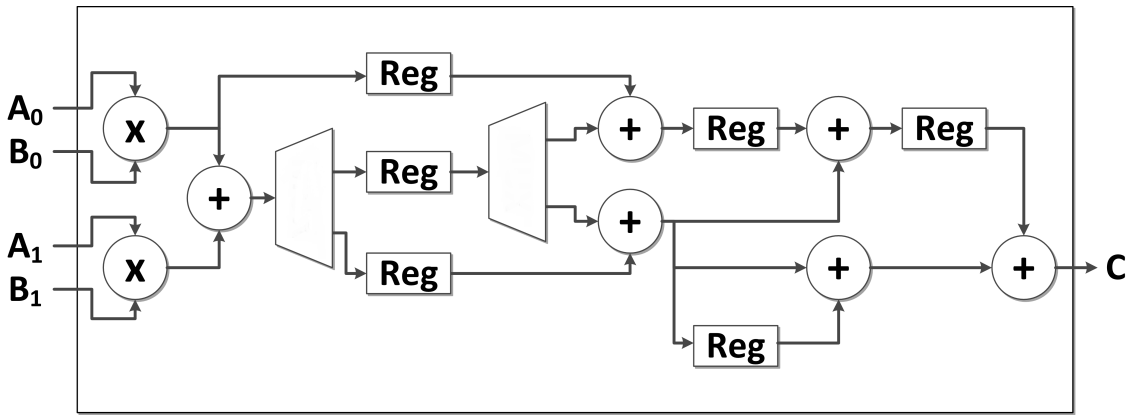

Figure 4.3: Standard HLS compute logic: cols pipelined.

Standard	*Factor*	*Mult*	*Add/Sub*	32 *Bit Reg*	8 *Bit Mux*	*Shift Reg*
No A.C.	*N.A.*	1	1	7	0	9
Product Pipelined	*N.A.*	1	1	7	1	10
Cols Pipelined	*N.A.*	2	6	18	36	18
Rows Pipelined	*N.A.*	16	20	128	0	144
Product Unrolled	16	2	6	18	40	18
Cols Unrolled	16	32	35	252	295	288
Rows Unrolled	2	182	202	1311	316	1638
Rows Unrolled	4	506	493	3519	378	4600

Table 4.1: Standard HLS component utilization.

Cols loop however, 8 of the two multipliers with shared adders were generated, as was originally expected.

4.2 Strassen Implementation

Unlike the standard design, the Strassen implementation was not based directly on an existing software implementation. Instead, the Strassen HLS design was approached with the desired hardware in mind rather than the sofware. The resulting code is presented in Figure 4.2.

for $i = 0 \rightarrow 1)$ **do** ▷ Outer
  for $j = 0 \rightarrow 1$ **do** ▷ Mid
    for $k = 0 \rightarrow 1$ **do** ▷ Inner

$$A' = A_{2i:2i+1,k:k+1}$$
$$B' = B_{2k:2k+1,j:j+1}$$

$$S_1 = (A'_{11} + A'_{22}) \times (B'_{11} + B'_{22})$$
$$S_2 = (A'_{21} + A'_{22}) \times B'_{11}$$
$$S_3 = A'_{11} \times (B_{12} - B'_{22})$$
$$S_4 = A'_{22} \times (B_{21} - B'_{12})$$
$$S_5 = (A'_{11} + A'_{12}) \times B'_{22}$$
$$S_6 = (A'_{21} - A'_{11}) \times (B'_{11} + B'_{12})$$
$$S_7 = (A'_{12} - A'_{22}) \times (B'_{21} + B'_{22})$$

$$C'_{11} = C'_{11} + S_1 + S_4 - S_5 + S_7$$
$$C'_{12} = C'_{12} + S_3 + S_5$$
$$C'_{21} = C'_{21} + S_2 + S_4$$
$$C'_{22} = C'_{22} + S_1 - S_2 + S_3 + S_6$$

    end for
    $C_{2i:2i+1,2j:2j+1} = C_{2i:2i+1,2j:2j+1} + C'$
  end for
end for

Figure 4.4: Strassen matrix multiply algorithm implementation.

The two inner-most for-loops work together to create the BEs of the hardware design. The outer-most exists to provide the functionality of the multiplexer in 3.3, that is, to provide two 2×2 matrices in series to be operated on.

Disregarding the outer loop, the trip count of the inner loop is equivalent to 4. Thus it is easy to see that when these two loops are fully unrolled the design should be equivalent (at least in terms of hardware components)

to the custom design. However, the HLS design has the advantage of being able to unroll the outer-most loop. In theory this would utilize more hardware resources than the custom design and provide a performance advantage.

As with the standard algorithm, various versions of the Strassen algorithm were examined. The first implementation tested was the HLS design with no architecture control added. The second was the design with the bottom level for-loop ($Inner$) pipelined. This was followed by the pipelining of the $Middle$ and $Outer$ loops. The next stage of improvement was unrolling each of the loops presented in the source code. As before, each of the loops (beginning with $Inner$) were successively unrolled for a total of three additional implementations. Each of the loops was fully unrolled, as the resource consumption was not as high as in the standard algorithm implementation. Figures 4.5 - 4.6 show two of the variations in RTL generated from the Strassen HLS design. Table 4.2 displays the component usage for each of the different HLS Strassen implementations.

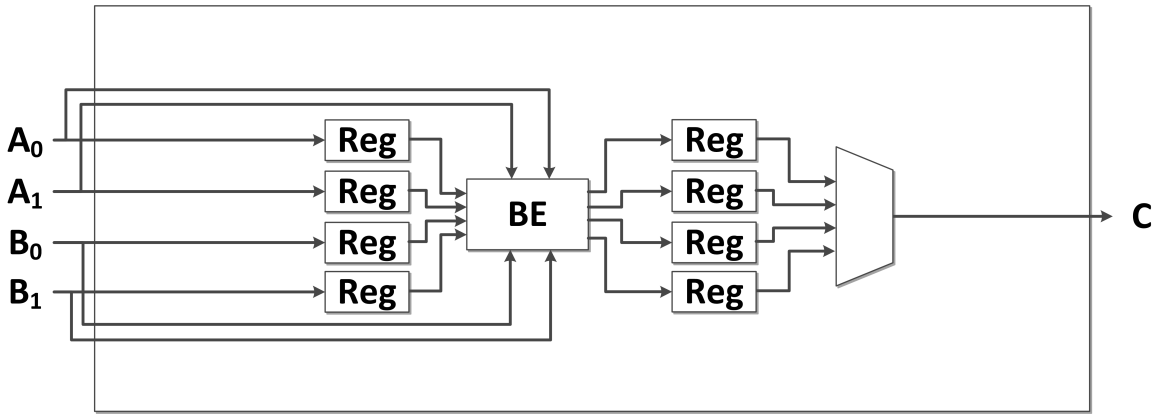

Figure 4.5: Strassen HLS compute logic: no architecture control.

As mentioned above, the BE computes the matrix multiplication for a 2x2 matrices. The loop bounds were set for a 8x8 matrix, and so unrolling the *Inner* loop produced two BEs and unrolling the *Mid* loop produced four BEs as shown in Figure 4.6. Additionally, since in this design more results are being calculated in parallel, the HLS tool generated a design with a second output port to allow for this increased bandwidth to be written out to memory. Pipelining the *Outer* loop just added pipelining to the unrolled

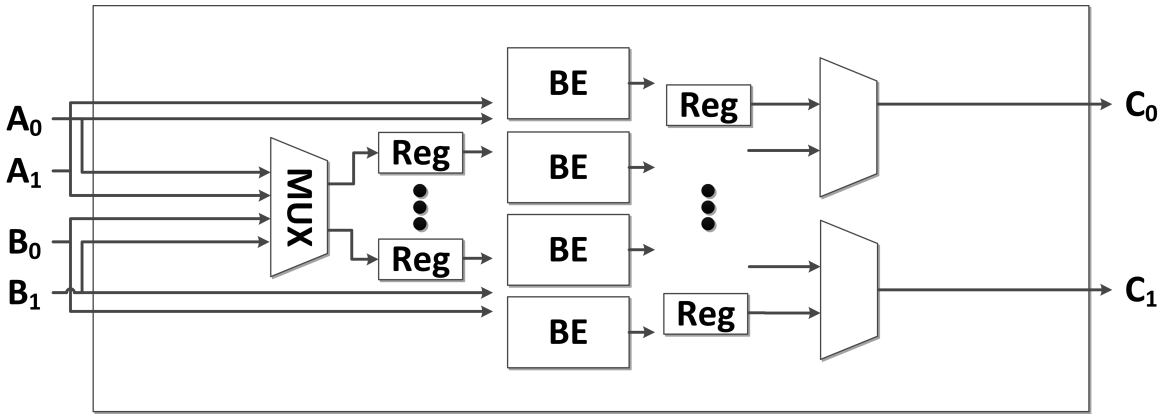

Figure 4.6: Strassen HLS compute logic: mid unrolled.

Strassen	*Mult*	*Add/Sub*	32 *Bit Reg*	32 *Bit Mux*	4 *Bit Mux*
No A.C.	7	22	54	5	16
Inner Pipelined	7	21	63	6	20
Middle Pipelined	14	40	111	5	20
Outer Pipelined	28	68	197	10	30
Inner Unrolled	14	40	102	5	25
Middle Unrolled	28	70	193	8	34
Outer Unrolled	56	120	367	22	2

Table 4.2: Strassen HLS component utilization.

Mid and *Inner* loops.

4.3 Sparse Implementation

For the sparse matrices implementation, A and B were input to the hardware function in compressed sparse row and compressed sparse column form respectively. The code consisted of 3 nested for loops, as with other implementations. It is presented in Figure 4.3.

This algorithm multiplies each non-zero element in a row of A with every non-zero element in a column of B and then repeat that process for every row and columnn of the matrix. Thus the top for-loop was iterated for each row in A. Given that the chosen block size was 16, the top loop was iterated 16 times. The middle for-loop needed to be iterated for each non-zero element in a given row of A. This value was obtained by calculating the

```
for i = 0 → rows(A) do                                          ▷ Top
    for j = row_A[i] → row_A[i + 1] do                            ▷ Mid1
        for k = 0 → cols(B) do                                      ▷ Mid2
            for m = col_B[k] → col_B[k + 1] do                        ▷ Bottom
                if col_A[j] == row_B[m] then
                    C_{i,k} = C_{i,k} + val_A[j] × val_B[m]
                end if
            end for
        end for
    end for
end for
```

Figure 4.7: Sparse matrix multiply algorithm implementation.

difference between sequential elements in the row pointer array of matrix A. The bottom for-loop needed to be repeated for each non-zero element in a particular column of B. This value was found by subtracting the values of adjacent elements in the column pointer array of matrix B.

The HLS sparse matrices implementation differed from the other HLS designs in that the middle and bottom loops iterated for a number of times based on an input. Thus the number of iterations for the two were variable. This prevented directives such as unroll and pipeline from having any effect on the performance of the implementation. Figure 4.8 shows the RTL from the HLS sparse matrix multiplier. Table 4.3 provides more details as to the actual hardware utilized.

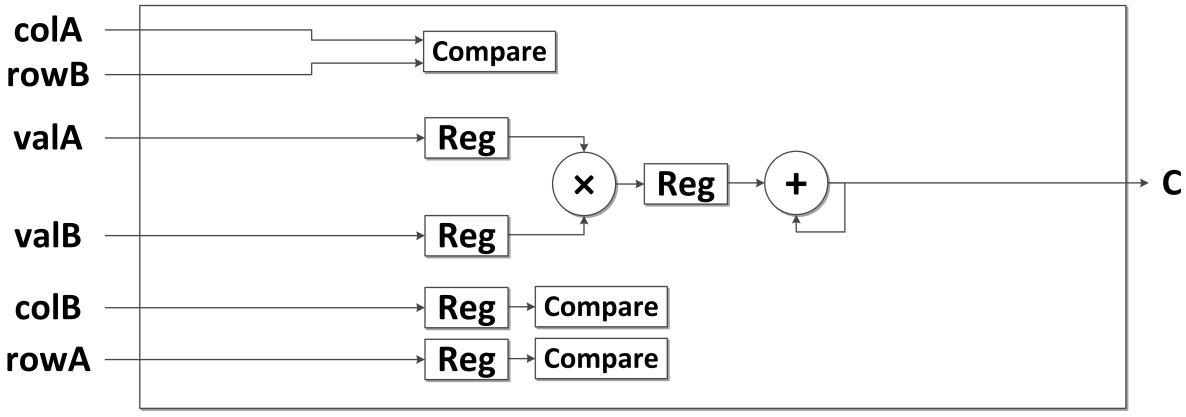

Figure 4.8: Sparse HLS compute logic.

$Sparse$	32 $Bit\ Mult$	32 $Bit\ Add$	32 $Bit\ Adder$	32 $Bit\ Mux$	32 $Bit\ Comp$	4 $Bit\ Mux$
$No\ Directives$	1	1	73	1	32	44

Table 4.3: Sparse HLS component utilization.

The sparse algorithm is very different from the other two algorithms in that the bounds on the loops are non-deterministic. The bounds depend on the sparsity and distribution of non-zero elements in the matrix. However, the *Top* loop is bounded to the number of rows in the matrix, and so this loop is able to be optimized in the HLS tool. The design with no optimizations is shown in Figure 4.8. The generated design has at its core a multiplier and an adder just like the standard algorithm but with extra inputs for the rows and columns to determine which elements to operate on. Unfortunately, due to the complexity of this design the HLS tool was not able to parallelize the algorithm in a beneficial way. As such, only the architecture diagram with no architecture control is shown here.

Chapter 5

System Design

5.1 Overview

Figure 5.1 shows the top level design for implementing the hardware accelerators.

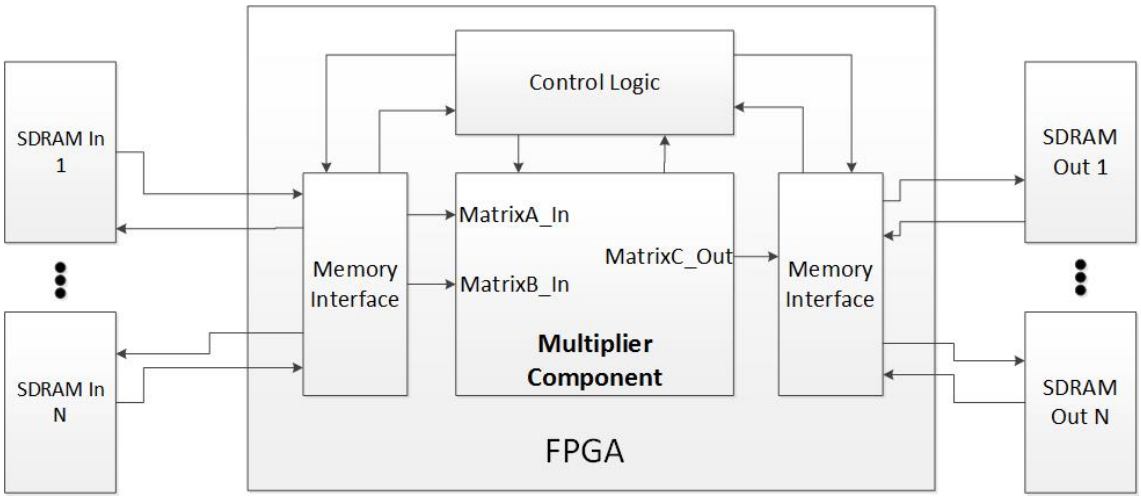

Figure 5.1: Design of the system from a top level perspective.

In order to provide for additional read and write buffering of the built-in memories, ping-pong buffers were utilized within the matrix multiplier component as is displayed in Figure 5.2. Each ping-pong buffer consisted of 16 distinct Block RAM elements which allowed for 16 simultaneous transfers (8 read and 8 write).

A standard DDR SDRAM memory with a 32 bit read/write width and clock speed of 400 MHz was considered for use in the proposed system. This device could provide $2 \times 32 = 64$ bits every clock cycle, or 64 bits every

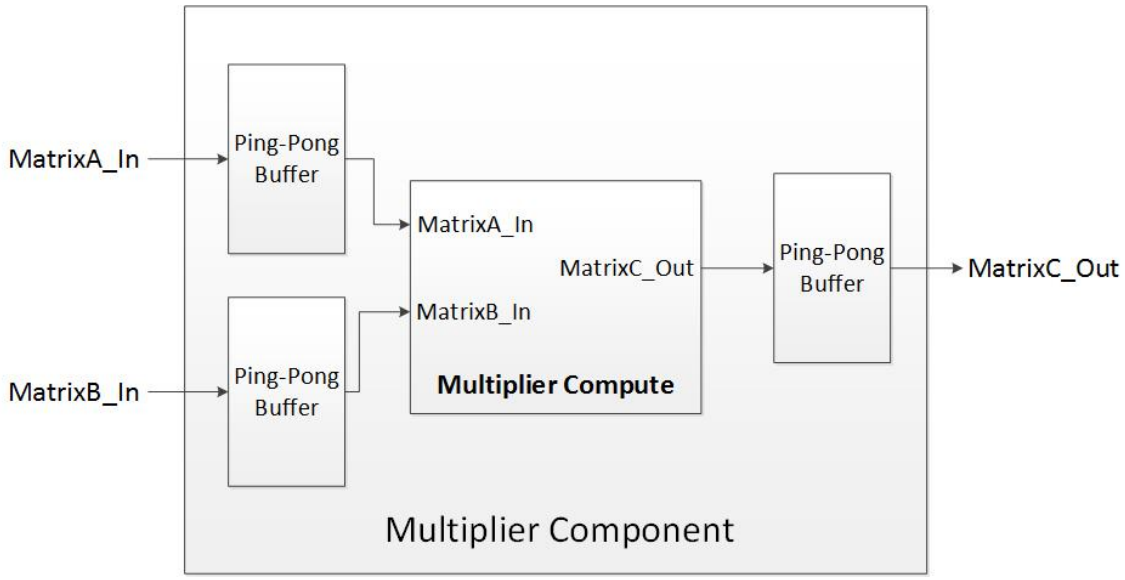

Figure 5.2: Design of the multiplier component, including the ping-pong buffers used increase throughput.

$(1/(400\times10^6)) = 2.5$ ns. Given that each of the accelerators performed calculations on 32 bit operands, this data was better expressed as 2 operands every 2.5 ns or 1 operand every 1.25 ns.

If an implementation required more operands per compute logic clock cycle then could be provided by a single memory component than a different approach other than connecting straight to external memory needed to be taken.

A pipeline was developed in order to satisfy those implementations that required large amounts of memory bandwidth. The first stage of the pipeline consisted of writing sub-matrices of size 64×64 to the built-in memories of the FPGA, located within the input ping-pong buffers of the multiplier component. The final stage of the top-level pipeline stored the resulting 64×64 matrix to external memories.

The second stage of the pipeline was the multiplication of the 64×64 matrices. This stage was divided into three separate sub-stages. The first sub-stage read two 16×16 matrices from built-in memory into internal buffers located within the multiplier compute block. The second sub-stage performed the multiplication of these matrices and stored the result into an internal buffer. The final sub-stage wrote the result buffer into the built-in

memory located in the output ping-pong buffer. Figure 5.3 shows the design of the pipeline.

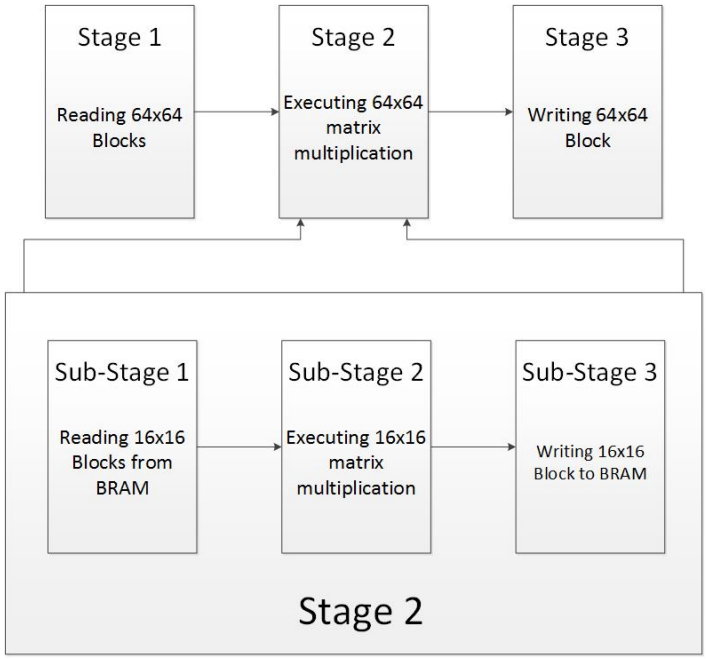

Figure 5.3: Pipeline designed to meet the high memory bandwith needs of the various matrix multiplier components.

Since the periods of the first and third stages of the pipeline could be adjusted by adding or removing memory elements the second stage was the limiting factor of the design. Within the second stage, the second sub-stage was dependent on the design that was being implemented. However, the periods of the first and third sub-stages were static regardless of choice of design.

As previously mentioned, a total of 8 elements could be read from/written to the built-in memory. A total of $16{\times}16 = 256$ operands needed to be read/written to/from built-in memory in a single cycle. Given a 2 cycle delay due to necessary control signals the latency of the first and third sub-stages was calculated as $2 + (256/8) = 34$ cycles. If the required cycles for the second sub-stage were lower than $2 + (256/8) = 34$ then the first/third sub-stage would be the limiting stage and the cycles for the sub-stage would be equal to 34. If the required cycles for the second sub-stage were greater than 34 then the number of cycles for the sub-stage would be that value. In

order to complete the 64×64 matrix multiplication $(64/16)^3 = 64$ iterations through the second stage of the pipeline are required. Given these constraints, total number of cycles required for the second stage was calculated as presented in Equation 5.1.

$$\#\,of\,Cycles\;=\;(\#\,of\,Iterations+(\#\,of\,SubStages-1))\times Cycles\,of\,SubStage \tag{5.1}$$

Since first stage of the described pipeline required a transfer of 2 64×64 matrices(A and B) the total number of operands that needed to be processed was $2\times256\times256 = 8192$. Given that a single SDRAM element can provide a single operand every 1.25 ns, one SDRAM component can provide the requisite number of operands in $8192\times1.25 = 10240$ ns. A pair of SDRAM elements working in parallel can write the operands in $10240/2 = 5120$ ns. The third stage of the pipeline required the transfer of only a single 64×64 matrix. Thus 1 SDRAM completes the operation in $4096 \times 1.25 = 5120$ ns or 2 SDRAMs in 2560 ns.

5.2 Pipeline Calculations

5.2.1 Standard Implementations

The compute logic for the custom standard design reads one row and one column of operands each cycle. The maximum clock frequency of the compute logic was found to be 213 MHz after implementation. At this speed each input required 16 operands every $(1/(213\times10^6)) = 4.69$ ns or 1 operand every $.29$ ns. Given that reading directly could only provide an operand every 1.25 ns it was clear that the standard custom design required the buffering capability of the pipeline.

As previously mentioned, the latency of the pipeline depended on the second sub-stage. Reading the entire 16×16 matrices required $(256/16) = 16$ cycles. Given that the longest sequence of adders and multipliers for the custom standard implementation was only 1 and 1, and the recommended latencies of the multiplier and adder IP cores were 6 and 2 respectively, the

number of cycles for the second sub-stage was calculated as $6+2+16 = 24$ cycles. Recall that the latency of the first and second sub-stages was 34 cycles. Thus the latency of the sub-stage was limited by the first/third sub-stages and was 34 cycles. The final cycle count for the second pipeline stage was then calculated as $(64 + (3 - 1))\times 34 = 2244$ cycles.

The period of the second stage of the pipeline was calculated as $2244\times (1/(213\times 10^6)) = 10535$ ns. Recall that utilizing a single SDRAM component for the first stage of the pipeline results in a period of 10240 ns. Thus in order to meet the period of the second stage of the pipeline only a single SDRAM to be used. The third stage period value of 5120 ns with a single SDRAM met the standard set forth by the second stage was therefore not a bottleneck in the design.

The best case maximum clock frequency obtained for any of the HLS standard matrix multipliers was 266 MHz. Thus the implementation required 4 operands (2 for A and 2 for B) every 3.76 ns or 1 operand every $.94$ ns. A single SDRAM component provided a single operand every 1.25 ns without any buffering. Thus 2 SDRAMs could be used to provide the operands to inputs A and B and implementation of the pipeline was not necessary.

5.2.2 Strassen Implementations

The custom Strassen implementation required 8 operands from both A and B each cycle. Using the maximum clock frequency (107MHz) the number of operands required for a single input was calculated as 8 operand every $(1/(107\times 10^6)) = 9.36$ ns or 1 operand every $.58$ ns. Thus the use of the pipeline was necessary.

As with the standard custom implementation, the cycles of the second sub-stage needed to be determined in order to determine the latency of the second pipeline stage. To complete the 16×16 matrix multiplication within sub-stage 2 $(16/4)^3 = 64$ iterations through the Strassen 4×4 multiplier were required. The longest elementary adder/multiplier chain through the Strassen custom implementation consisted of 4 additions/subtractions and 1

multiplications. Given the latencies of the IP cores recommended from Xilinx, this totaled to $4\times2+6=14$ cycles. One iteration through the Strassen 4×4 multiplier required 2 cycles worth of operands to complete. Thus the total number of cycles required to complete a 16×16 matrix multiplication was equivalent to $2\times64+14=142$ cycles. Given this value, the cycle count for the second pipeline stage was calculated as $(64+(3-1))\times142=9372$ cycles.

The number of cycles was used with the maximum clock frequency to calculate the period of the second stage of the pipeline as $9372\times(1/(107\times10^6))=87,589$ ns. The first stage of the pipeline has a period of 10240 ns or 5120 ns when 1 or 2 SDRAMs are used respectively. Since the period with only 1 component falls well beneath the value for the second stage of the pipeline only a single SDRAM needed to be used in the first stage. The third stage of the pipeline also easily meets the value set by the second stage with a single SDRAM.

The best case maximum clock frequency obtained for any of the HLS Strassen matrix multipliers was 180 MHz. The Strassen HLS multiplier required 4 operands every $1/(180\times10^6)=5.54$ ns or 1 operand every 1.385 ns. Therefore a single SDRAM component capable of providing a 32 bit operand every 1.25 ns could satisfy both inputs A and B of the Strassen HLS implementation.

5.2.3 Sparse Implementations

A single SDRAM provides 32 bits every 1.25 ns. The maximum clock frequency for any of the custom sparse designs was 341 MHz. Unlike the other implementations, indices also needed to be read from memory. The maximum value for a index was 16, as the implementation was designed to operate on 16×16 matrices. Therefore each index needed to be 4 bits wide ($2^4=16$ possible values). This meant that a total of 24 bits needed to be read from memory every $(1/(341\times10^6))=2.93$ ns. Thus in order to satisfy the requirements, 3 distinct SDRAMs need to be utilized for the custom sparse implementation, 1 for each of the input matrices and 1 to handle the indices.

The maximum clock frequency for the HLS sparse implementation was found to be 121 MHZ. Given that the design required 4 operands every $1/(165\times10^6) = 6.04$ ns and that single SDRAM provided an operand every 1.25 ns only 1 SDRAM was required for this implementation with no pipeline implementation necessary.

Chapter 6

Results

6.1 Standard Results

The hardware resources consumed by each of the implemented standard designs are presented alongside the performance speedup compared to the software design in Table 6.1.

Pipelining the innermost loop $Product$ resulted in a speedup three times greater than that of the non-optimized design at the cost of very few resources. Likewise, pipelining $Cols$ yielded a speedup five times greater than that of the $Product$ pipelined implementation. On the contrary, pipelining the outermost loop $Rows$ gave a noticeable increase in resource consumption with no improvement in speedup. This is due to the decrease in maximum clock frequency associated with the increase in hardware utilization of the design.

Unrolling $Product$ resulted in an improvement in speedup by a factor of seven with a minimal increase in resource consumption. However, the performance to resource compsumption ratio greatly decreases with additional unrolls. Unrolling $Cols$ decreased doubled the speedup of the design but at a cost of using five times the number of DSPs. This trend continued as an unroll of the $Rows$ loop unrolled with a factor of 2 yielded a slightly improves speedup but a DSP usage of 27 percent. Again this is due to the much lower maximum clock frequency of the larger designs.

Given these results it is clear that applying architecure control to loops within the standard multiplier had diminishing returns in the standard HLS designs. Though pipelining and unrolling outer loops decreased the latency (number of clock cycles) of the matrix multiplication computation, it also

Optimization	**Resources** [Total]			**Speedup**
	LUTs [297760]	**FFs** [595520]	**DSPs** [2016]	
None	1%	1%	1%	0.2x
Product Pipelined	1%	1%	1%	0.6x
Product Unrolled	1%	1%	1%	3.2x
Cols Pipelined	1%	1%	1%	3.2x
Cols Unrolled	1%	1%	5%	1.5x
Rows Pipelined	1%	1%	2%	2.7x
Rows Unrolled - 2	3%	2%	27%	3.1x
Rows Unrolled - 4	7%	5%	75%	4.8x
Custom	1%	1%	13%	50.6x

Table 6.1: Percent of resources utilized and speedup compared to software implementation of standard algorithm.

greatly decreased the clock frequency at which the designs could operate. This negated much of the performance gain from the decreased latency. This is due to the inefficiencies associated with the HLS tools in controlling large designs. When the designs are small the HLS tools can easily generate a state machine that controls data flow fairly efficiently. However, when designs are larger the control logic auto-generated from the HLS tools is unable to handle the data efficiently, causing the much lower maximum clock frequencies.

The HLS design with the largest speedup compared to the software design was the Rows Unrolled - 4 implementation which obtained a speedup of 4.8. The custom standard design achieved a speedup of 50.6, over 10 times greater than that of the Rows Unrolled - 4 implementation. In addition, the custom design utilized fewer resources than the Rows Unrolled - 4 implementation. Perhaps the most notable discrepancy between the two designs lays in the fact that the HLS design only reads 2 elements from each input matrix into its buffers simultaneously. Recall that in the ping-pong buffers used in the custom design a total of 8 simultaneous reads were possible. This difference meant that the custom implementation could read data 4 times as fast as the HLS implementation which contributed greatly to the inability of the HLS implementation to compete in terms of performance.

Optimization	**Resources** [Total]			**Speedup**
	LUTs [297760]	**FFs** [595520]	**DSPs** [2016]	
None	1%	1%	1%	0.4x
Inner Pipelined	1%	1%	1%	0.5x
Inner Unrolled	1%	1%	2%	1.0x
Mid Pipelined	1%	1%	2%	2.0x
Mid Unrolled	1%	1%	4%	1.6x
Outer Pipelined	1%	1%	4%	2.0x
Outer Unrolled	1%	1%	8%	2.9x
Custom	1%	1%	6%	4.8x

Table 6.2: Percent of resources utilized and speedup compared to software implementation of Strassen algorithm.

6.2 Strassen Results

The hardware resources consumed by each of the Strassen implementations are presented alongside the speedup over the software implementation in Table 6.2.

The initial pipeling of the innermost loop did not result in a vast improvement in speedup over the unoptimized design. Pipelining the $Middle$ loop however yielded a 5 times improvement over the $Inner$ pipelined implementation. Contrarily, pipelining the loop $Outer$ resulted in a notable increase in resource consumption with little improvement in speedup. This is due to simply the nature of the algorithm and how little it is impacted by pipelining. Pipelining the $Middle$ loop only provided a large increase in speedup because by default it unrolled the loop beneath it, $Inner$.

Unrolling $Inner$ doubled the amount of DSP slices consumed and increased the speedup by a factor of 2. This trend continued as the unrolling of both the $Middle$ and yielded a doubling of consumed resources and approximate doubling of speedup. This is due to the decrease in latency associated with performing more the computation in parallel and the the steady maximum clock frequency across designs.

Unlike the standard HLS implementation, successive unrolls of the Strassen implementation did not yield diminishing returns with regards to performance. Each unroll provided a steady linear gain in performance. Comparing the resource utiliziation between the two algorithm implementations, it

is clear that even the largest of Strassen implementations were still small compared to the large standard implementations. Thus the Strassens designs were small enough for the control logic auto-generated from the HLS tools to be efficient, resulting in a fairly constant maximum clock frequency across the different implementations. This meant that the decreased latency resulting from unrolling the algorithm directly correlated to an increase in speedup of the computation.

The $Middle$ unrolled implementation represents the attempt at replicating the exact hardware designed for the Strassen custom implementation through use of the HLS tools. Being able to compare the custom implementation to an HLS counterpart that utilized the same number of elementary multipliers gave a unique opportunity to examine the differences in design. The first thing to note is that the the custom design uses 1.5 times as many DSP slices as the HLS design. This is due to the optimizations made within the Xilinx Multiplier IP core of the custom design. The same problem with the number of simultaneous data reads that existed with the standard HLS design also exists with the Strassen HLS design. The ability of reading 2 operands simultaneously simply does not compete with the ability of the custom design to load 8 operands into its buffer simultaneously.

6.3 Sparse Results

The speedup of the sparse matrix multiplier implementations over the software design are presented alongside hardware resource usage in Table 6.3.

Optimization	**Resources** [Total]			**Speedup** [Density]		
	LUTs [297760]	**FFs** [595520]	**DSPs** [2016]	[30%]	[20%]	[10%]
None	1%	1%	1%	1.2x	1.0x	0.6x
Top Pipelined	1%	1%	1%	0.9x	0.8x	0.5x
Top Unrolled	12%	4%	1%	0.5x	0.4x	0.2x
Custom PE - 4	1%	1%	1%	223.9x	140.4x	56.7x
Custom PE - 8	1%	1%	2%	240.0x	132.0x	43.8x

Table 6.3: Percent of resources utilized and speedup compared to software implementation of sparse algorithm.

Unlike the standard and Strassen algorithms, applying optimizations within

the HLS tool did not provide any performance boost over the non-optimized design. In fact, each of the tested optimized designs actually reduced the speedup nwhen compared to the non-optimized design. This is due to the additional control logic necessary to implement the optimizations. As previously mentioned, control logic is a weakness of the HLS tools. The sparse algorithm, with its non-deterministic for-loops, is the most control intensive of the algorithms implemented. The only HLS design that managed a speedup greater than 1.0 was the non-optimized design in the case of matrix densities of 30. In general, as matrix density decreased the HLS designs became less efficient.

The custom design performed better than the HLS designs in all test cases. The systolic array structure and custom control logic meant that the custom design was able to utilize additional processing elements and efficiently distribute the workload in a parallel fashion. Given these results, HLS tools are not well suited for algorithms with non-deterministic loop bounds.

Chapter 7

Design Time Comparison

The performance and design time of implementing each of the three matrix multiplication algorithms in software, HLS, and custom is shown in Figure 7.1.

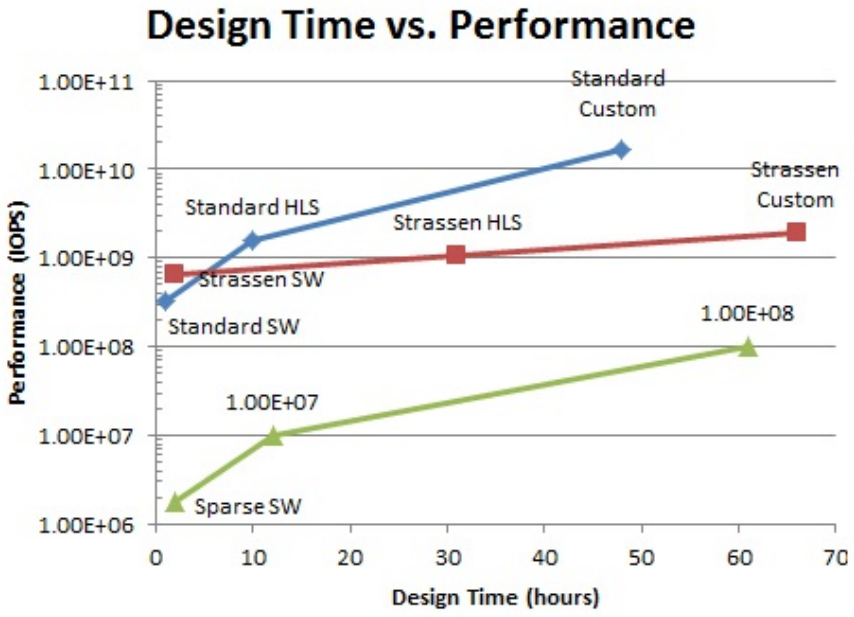

Figure 7.1: Design time of matrix multiplication algorithms and their various implementations.

The performance was measured through IOPS (integer operations per second) and the design time was measured in hours required to complete

each design. There is a clear pattern that can be established across the different algorithms. In each case the software implementation had the lowest design time and the custom implementation had the longest design time, with the HLS implementation falling in the middle. The degree to which the different implementations varied in design time depended on the algorithm. In the cases of the standard and sparse algorithms, where the HLS source codes were ported directly over from established software implementations, the gap was very large. This was due to the ease in transitioning from a functional software design to an HLS design.

The Strassen HLS implementation was designed to mimic the architecture of the developed Strassen custom design. The Strassen HLS design took significantly longer than that of the other two algorithms due to the fact that it wasn't ported from an existing software design. However, the Strassen HLS design performs closest to its custom implementation when compared to the other two algorithms. Thus the additional design time of the Strassen HLS implementation yielded a net gain in performance.

The differing design times of the custom designs depended largely on the nature of each algorithm and the complexity design. The standard custom design took the least amount of time due to the fact that it consisted of large numbers of elementary multipliers and adders connected in parallel. The Strassen custom design required large elementary operation chains in order to form the intermediary matrices necessary for the algorithm. In addition, the input buses needed to be multiplexed in order to switch between different source matrices. The sparse custom implementation followed a systolic array structure that needed to be able to easilt toggle between different numbers of processing elements. In terms of complexity, this design fell between the fairly straightforward standard custom design but far short of the more complicated Strassen custom design. As such, the design time for the sparse custom design fell in between that of the other two algorithms.

Chapter 8

Combined Custom/HLS Design Flow

It is clear that while custom designs outperform HLS designs, they also take significantly longer to design. In general, this performance gap can be bridged by applying optimizations to the HLS designs, though cases exist (such as with the sparse algorithm case) in which the optimizations do not improve performance In addition, approaching HLS design from an angle alternative to porting over existing software code (such as was done with the Strassen algoritm) can also yield increased performance. With these conclusions in mind, a design flow such as presented in Figure 8.1 is recommended.

The first step in the design flow would be to research the application and determine if there is any established software implementation that could be ported into the HLS tool. Next would be performing a check to make certain that the application is suitable for implementation using the HLS tool by checking for things such as non-deterministic loops. If it is not, then a custom design is necessary and the designer can move directly to following a design flow similar to that presented in 2.6. If no pitfalls are found and the application is deemed suitable for HLS implementation than the designer can move to following a HLS design flow similar to that shown in 2.7.

After running the HLS tool, the decision must be made as to whether or not the performance needs of the application have been met. If they have not, the designer must determine whether or not performance gains can be made through using optimizations such as loop-unrolling and pipeling within the HLS tool. If so, then the respective directives should be added to the design and the HLS design process repeated. If a point is reached

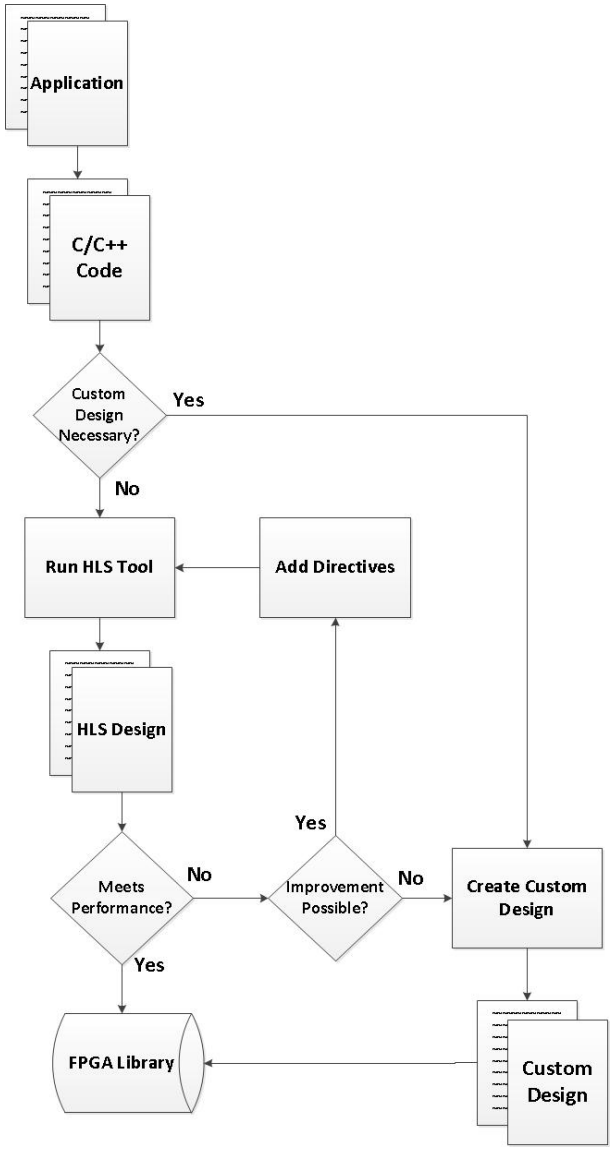

Figure 8.1: Example of an efficient design flow for developing applications on the FPGA.

where the design still does not meet the performance needs of the application and the optimizations within the HLS tool have been exhausted then the designer will need to develop a custom design. When a design meets the requirements set forth by the application it is stored in a library for future use.

Chapter 9

Conclusions

Design time is a huge barrier to utilizing FPGAs in heterogeneous systems. For many applications, obtaining maximum performance is not a requirement. This paper has shown that speedup over traditional software implementations is achievable with minimal design time using HLS tools for several different multiplication algorithms. The performance gap between HLS and custom designs can be lessened by optimizing HLS designs. A design flow has been presented that, given the performance needs of an application, can greatly reduce the design time of an FPGA implementation. As HLS tools improve in both their usability and performance, the number of applications that require custom applications will decrease. This makes FPGAs a significantly more attractive option for implementation within a heterogeneous system.

Bibliography

[1] Nathan Bell and Michael Garland. Implementing sparse-matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009.

[2] Ignacio Bravi, Jimenez Pedro, Jose Luis Lazaro, Jose de las Heras, and Alfredo Gardel. Different proposals to matrix multiplication based on FPGAs. In *IEEE International Symposium on Industrial Electronics*, 2007.

[3] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. Strassen's algorithm for matrix multiplication. In *Introduction to Algorithms*, 2001.

[4] Wim Meeus, Kristof Van Beck, and Toon Goedeme. An overview of today's high-level synthesis tools. In *Springer Science and Business Media*, 2012.

[5] Ioannis Sotiropoulos and Ioannis Papaefstathiou. A fast parallel matrix multiplication reconfigurable unit utilized in face recognition systems. In *International Journal of Computer Applications*, 2009.

[6] Prasanna Sundararajan. High performance computing using FPGAs. In *White Paper: FPGAs*, 2010.

[7] Xilinx. FPGA design overview. In *ISE*, 2008.

[8] Xilinx. Virtex-6 FPGA configurable logic block. In *User Guide*, 2012.

[9] Xilinx. Virtex-6 FPGA DSP48E1 slice. In *User Guide*, 2012.

[10] Xilinx. Virtex-6 FPGAs memory interface solutions. In *User Guide*, 2012.

[11] Colin Yu Lin, Zheng Zhang, Ngai Wong, and Hayden Kwok-Hay So. Design space exploration for sparse matrix-matrix multiplication in FPGAs. In *International Convference on Field-Programmable Technology*, 2010.

[12] Ling Zhuo and Viktor Prasanna. Design tradeoffs for BLAS operations on reconfigurable hardware. In *International Conference on Parallel Processing*, 2005.

[13] Ling Zhuo and Viktor Prasanna. High performance designs for linear algebra operations on reconfigurable hardware. In *Computers, IEEE Transactions*, 2009.